Formalizing graphical notations

Thesis

Formalizing Graphical Notations

Submitted by:

William Godwin, MA (Cantab.), Dip. Lingu., Cert. Ed.

to the Open University

as a thesis for the degree of Doctor of Philosophy

in the discipline of Computing Science

Date of submission:

29th January 1998

I certify that the work and ideas presented in this thesis are wholly my own, and all other sources are clearly referenced in the text.

W. H. Godwin

AUTHOR'S NUMBER: M7045766
DATE OF SUBMISSION: 30 JANUARY 1998
DATE OF AWARD: 7 JULY 1998

ALL MISSING
PAGES ARE
BLANK
IN
ORIGINAL

To my parents Ida and Edgar, my brother Jim and my wife Jennifer

Contents

Acknowledgements

The development of ideas in this thesis has been achieved without expert assistance in the specialist areas covered. This research has been carried out without benefit of financial assistance or external funding.

I would like to thank my supervisors Professor Darrel Ince and Dr. Sohrab Saadat for their consistent encouragement and advice, and all those who have read and commented on my work, in particular Dr. Stella Mills.

I would also like to thank the Cheltenham and Gloucester College of Higher Education for circumstancial support for the duration of the work, which has made the research possible. I am indebted to the libraries of Bath University and the Science and Engineering library at Bristol University, whose facilities I have regularly consulted.

Finally I would like to thank my wife Jennifer, without whose immense patience, encouragement and practical support this thesis would not have reached fruition.

Thesis Abstract

The thesis describes research into graphical notations for software engineering, with a principal interest in ways of formalizing them. The research seeks to provide a theoretical basis that will help in designing both notations and the software tools that process them.

The work starts from a survey of literature on notation, followed by a review of techniques for formal description and for computational handling of notations. The survey concentrates on collecting views of the benefits and the problems attending notation use in software development; the review covers picture description languages, grammars and tools such as generic editors and visual programming environments. The main problem of notation is found to be a lack of any coherent, rigorous description methods. The current approaches to this problem are analysed as lacking in consensus on syntax specification and also lacking a clear focus on a defined concept of notated expression.

To address these deficiencies, the thesis embarks upon an exploration of semiotic, linguistic and logical theory; this culminates in a proposed formalization of semiosis in notations, using categorial model theory as a mathematical foundation. An argument about the structure of sign-systems leads to an analysis of notation into a layered system of tractable theories, spanning the gap between expressive pictorial medium and subject domain. This notion of 'tectonic' theory aims to treat both diagrams and formulae together.

The research gives details of how syntactic structure can be *sketched* in a mathematical sense, with examples applying to software development diagrams, offering a new solution to the problem of notation specification. Based on these methods, the thesis discusses directions for resolving the harder problems of supporting notation design, processing and computer-aided generic editing. A number of future research areas are thereby opened up. For practical trial of the ideas, the work proceeds to the development and partial implementation of a system to aid the design of notations and editors. Finally the thesis is evaluated as a contribution to theory in an area which has not attracted a standard approach.

Chapter 1

Introduction

Abstract

Here we find the reasons for this research effort in the chosen topic of formalizing graphical notations. The research is motivated by a perceived need for flexible notational techniques in all aspects of software development. For a rigorous approach to development, notation processing must also be computer-aided. The objectives of the research therefore focus on formal specification for syntax and practical support for designing and using notation.

The proposed method of research is described as an applied mathematical study, starting from an informal discussion, that is inspired by a semiotic view of human processes of idea-sharing and problem-solving. The research is related to other work on visual language, diagrammatic reasoning and computational linguistics, but intends a fresh and fundamental approach.

Finally we find an overview of a thesis that applies and combines notions from a range of disciplines, in order to address the problems of describing, designing and processing formal graphical notations, particularly in relation to the tasks of developing computer systems.

Chapter 1.

Introduction

By relieving the mind of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. – Alfred North Whitehead (1948)

This opening chapter invites the reader to place themselves in the position of someone whose main interest lies in graphical expressions and diagrams that occur as technical notation within a design discipline such as software engineering. The text indicates the area of study, discusses methods of research, places the area in the context of other work and finally advertizes the content of the remaining chapters.

1.1 The Study of Notations

The first task is to introduce the subject area, to state broadly what aims and objectives the research is to pursue, and to outline reasons why the research is being undertaken.

This work, then, is a thesis about notations – or, to be more specific, systems of graphical symbols that are used fairly formally in mathematics, the sciences and technology. In particular, the study focuses on kinds of diagram common in the practices of software engineering, where problems of notation are often observed. Research into such technical notations tries to understand how the techniques and mechanisms available in the graphical medium make it possible for drawn expressions to carry information.

1.1.1 Aims and Objectives

The work aims to provide theoretical support for both the design and use of notations. It aims to give the designers of a notation a way to describe and depict its structure, and thereby help in constructing generic editing and processing tools for diagrams in software development. This will offer better computer assistance to notation users, allowing the procedures of editing to be guided by the 'pictorial-syntactic' structure of a notation.

The main objectives are firstly to establish an effective formal way for specifying the syntax of graphical expressions, and secondly to test the method practically, by building a prototype *notation design* tool. It is intended that the designed syntax specification itself be notated graphically.

1.1.2 Perspective

How should we regard expressions and notations? Drawn expressions are in one sense cognitive objects, and in another, physical objects; a notation belongs, however, to the group of people that uses it, and may be regarded as a cultural object, whose manifestations have cognitive and physical attributes.

Software development is after all a 'cultural' process, which is subject to structures of social convention within a community of practitioners. It is thereby a field in which sign-systems play a large role; being a technical activity, we might expect such *semiotic* processes to be within reach of mathematical analysis.

Such a formal analysis would also offer support in modelling human interaction with computers. Users ought to be able to *engage* with software (Laurel 1991), to feel involved as a participant in the domain of activity it represents to them. This they can do by internalizing its semiosis – learning the unconscious cognition needed to extract the meanings of images and interactions – provided that the developer has designed into the software a suitable 'semiotic capacity' to make this feasible. Developers themselves use computer aids in order to help create application systems. Whether notations are for developers or for end-users, they must have their 'rules of engagement', which this work endeavours to study.

1.1.3 Motivation

How will it be helpful, to formalize notations in software engineering? We find that diagramming methods have evolved informally in the software industry. By being adapted to the skills of current personnel, they provide flexible thinking tools that can represent structure in a variety of ways. Unfortunately notations are in the main not precisely defined and not rigorously usable in the development process; exactness is only enforced at the implementation stage, by the formality of program codes. By opening the way for computer-aided reasoning, formalized notation could help trap errors that may occur early in design and which can become costly if not discovered until the testing phase.

With regard to this difficulty, Formal Methods notations have been devised to support rigorous methods of software development; they are concise, unambiguous and they enable logical checking throughout development. They are, however, neither pictorial nor familiar to personnel, who

who perceive them to be difficult to learn and use. In this case the research might make it feasible to pictorialize and thereby simplify any work with formal notations.

The formalization of graphical notations promises a way to integrate the flexibility and familiarity of informal design practices with the rigorous standards of expression essential for verifiable and re-usable software. This research does not, however, seek to prescribe any particular form or method of using notations, but to find techniques for describing and supporting those forms that exist, even where these make no claim regarding formality.

Computer-aided tools for software development are generally designed for a restricted range of graphical notations, depending on the individual choices and programming techniques adopted by the tool maker. Few of them support mixing of notations from different methods. By way of contrast, in compiler design, systematic techniques apply across a range of textual languages, resting on mathematical syntactic theory. We might aim to make the same possible for diagrammatic syntax.

1.1.4 Application of this Thesis

Who will benefit? This work is intended to be applied in providing a standard for formal definition of notational structure, in building general-purpose tools for practitioners using software engineering notations, and in improving design of notation within a computer system interface, for novices and experts in any field. In recent years, the increased availability of graphical user interfaces, and the popular appeal of visual methods and programming tools are evidence for growing opportunities to apply this research.

1.2 Method of Research

How is the research to be conducted? Different disciplines practise different methods of researching; what is appropriate here?

For example in **science** the practice is to formulate hypotheses about certain phenomena and tests these by experiment or observation. In order to explain clusters of co-occurring phenomena, statements must be precise and logically consistent, though they may refer to entities that are not known to exist.

In the **humanities**, establishing a hypothesis requires the reaching of agreement on the explanatory usefulness of its concepts. The discussion of hypotheses helps to clarify intuition in an area where consensus is lacking. Without an agreed conceptual framework, scientifically testable hypotheses cannot be framed.

In pure **mathematics**, we can discuss any concept in a very precise manner, without regard to external validity. Mathematics justifies its constructed theories by their elegance, as well as relevance to other internal problems. In **applied** mathematics, we aim to develop an elegant theory and to form an *analogy* between its terms and the concepts of some external problem. Practical problems can present a considerable challenge to the seeker of elegant theories.

This thesis is conceived as a study in applied mathematics. The subject is the use of notations in software development. Its inspiration comes from a semiotic perspective on the human processes of problem-solving in developing computer systems. These ideas will be discussed informally, following the tradition in the humanities; from this discussion, concepts will be selected for more precise treatment. The mathematics will allow concepts to be developed formally, so that theoretical and experimental science can in the future make use of them.

1.3 Relation to Other Work

What other researches are relevant? Most work related to this topic is of very recent origin and is motivated by use of computers. Notational concerns are discussed in relation to specific *software design methods*, and sometimes more generally. Research into *visual programming languages* is an important new development. Many projects are concerned with building effective general support tools such as syntactic *diagram editors*, or compilers for visual languages. Cognitive and logical issues are the subject of active projects researching into *diagrammatic reasoning* and other aspects of *knowledge representation*. Techniques of *graph rewriting* have become an important area of study that is applied to generalized notions of grammar. Of less direct relevance are work on qualitative *spatial theories* and systems of *constructive logic* – which are applied to notation description and processing. More broadly, the thesis relates to the problem of *formalizing semiotics*, which has arisen out of philosophical logic traditions.

This variety of work is mostly not directed strictly at notation itself, but at concerns that overlap the topic in some way.

1.4 Overview of the Thesis

This thesis draws together notions in linguistics, logic, category theory and computing, and applies them to the description, design and processing of formal graphical notations for computer systems development.

Chapter 2 surveys the notation issues described in the literature, starting with a search for theories about notational design. It attempts to discover the roles that notations play and the factors that affect their design, especially in regard to formality and pictorial properties. It investigates the problems and weaknesses that are reported, and how these have been addressed.

The more formal approaches that have been tried are the subject of **Chapter 3**, which collects views on techniques and tools available for work with notations in the software development context. It reports on methods of formal representation that have been used in linguistics and applied with varying success to graphical language, and it reviews research into systems that offer editing or other processing of notation. The techniques and approaches are evaluated and their difficulties are analysed in order to formulate specific aims for this research.

Chapter 4 defines the chosen area of research and explores the problems identified, starting from the viewpoint of semiotic theory. The particular characteristics of notations as symbol systems are discussed, and an examination of linguistic researches gives evidence for a logic-based approach. The challenge is to provide a uniform method of description for notations as layered logical structures – a method that can support varied operations on notations. To meet this, an outline for a 'tectonic' theory of notation systems is presented.

The topic of pictorial syntax is the focus for **Chapter 5.** It describes a schematic notation (SIGN) proposed for syntax definition, based on the Theory of *Sketches* as a logical *institution*; these 'sketches' are mathematical objects that are presentations of theories. As the name implies, sketches lend themselves to graphical expression, which is in the manner of diagrams often used by category theorists. The method of definition with SIGN is explained and demonstrated in detail and applied to the case of Jackson's Structure Diagrams.

Other structural layers are dealt with in **Chapter 6**, with a view to supporting the use of SIGN in constructing generic diagram editors. The chapter suggests possible ways of applying the tectonic theory in depth, and addresses several topics which offer directions for further research.

Interpretive processes are considered and treated as systems of logical deduction; theories needed for geometric and graphical aspects of expressions are discussed. To support editing, mechanisms of rewriting are examined as ways of effecting changes to expressions.

Chapter 7 proposes designs for a Notation Design Assistant (AGENDA) based on the principles of chapter 6. It then presents a more limited prototype generic tool for development of notations and diagram editors, and discusses implementation choices. It reports on the work of partially implementing the prototype, highlighting what has been learned in its construction. To assist in designing a notation, the prototype allows a user to build the syntax, to assign pictorial-geometric realizations, and to develop a simple editor.

Chapter 8 concludes by summarizing and evaluating what has been achieved in relation to the objectives, the problems addressed and in comparison with other work. It looks forward to further developments of the approach.

Appendix A gives examples to show how symmetry in expressions may be formally described.

Appendix B treats an example of re-formulating a BNF grammar as a sketch.

Appendix C illustrates reasoning with sketches, and shows a strategy for displaying formal proofs, with a proof about logical properties of a simple rewrite rule.

Appendix D lists definitions of new or unfamiliar terms and concepts used in the thesis.

Appendix E briefly outlines an approach to formalizing graphical metaphors.

Appendix F lists selected Smalltalk classes and methods developed in the prototype tool, to illustrate the design.

Chapter 2

Survey of Problems

Abstract

We here find a survey of views and researches into notation usage. From a historical perspective, the survey shows that little explicit analysis of the topic exists before the advent of computers. Early writings by mathematicians link notational concerns to the development of formal logic, which includes the origins of the subject of semiotics. More recent attention is reported to the study of visual languages in computing and the use of diagrams as tools for logical reasoning.

Software engineering activities and respective notations are then classified, and the survey consults many views on notational needs and some assessments of performance. Analysis of reported problems and weaknesses of the notations shows that the varied needs have been satisfied in a piecemeal manner. Thus programming codes are formal but overly restricted; requirements analysis and systems analysis are served by informal diagramming techniques, unsupported by any clear formulation of the development process; formal specification and refinement methods have adopted much of the style of mathematical formulae, with little concern for ease of comprehension.

Overall there is seen to be little awareness of any theoretical principles governing notations. This general lack of support for notation design is a cause for concern – continuing development and invention of notations is called for in software engineering, because of changes brought about by new technologies and applications.

A summary of issues and difficulties raised by authors yields evidence that matters of notation design could be improved by formalization. Formal descriptions, it is argued, would bring precision to the many stages of software development and simplify the building of computer aids for notation. Suitable theory and science for notation structure would result in more reliable design principles and thus better decisions about styles and characteristics of notations used. Some possible programmes for research are suggested.

Chapter 2.

Survey of Problems

Confucius is often quoted as saying that a picture is worth ten thousand words – so please never draw one that isn't. – C.A.R. Hoare (1986)

This chapter is devoted to a survey of the literature relating to notation and its place in software development. Its purpose is to collect and analyse commentaries on the properties required of notations. The survey starts from an historical perspective, with reports on notation problems that originate in relevant topics of mathematics. An overview of our chosen context is then provided, briefly classifying software engineering activities and their notations – after which the survey consults many views on how notations perform in the practice of system development. The writers selected have reported on problems and weaknesses recognized, and on how these have been (or could be) addressed.

In all this material we seek to discover how important notations are, what roles they play, and what are the factors that affect their design. We look for stated principles of design, in particular opinions on formality and pictorial form. From these reports, we can analyse the issues raised and summarize the problems and concerns. What kinds of problem do people encounter when designing or using graphical notations?

2.1 The Study of Notation Systems

It is remarkable that there is no tradition of theoretical study dedicated to the specific subject of technical notations. In view of this lack, we instead take note of the reported views of early researchers in mathematical logic. In addition, we note that there is current of theoretical interest, in areas such as the use of diagrams as tools for logical reasoning.

This material provides us with a suitable perspective from which to appreciate the notational needs in the chosen context of software development.

2.1.1 Historical Material

It is appropriate to begin with a rare commentary on the concerns of past mathematicians who wished to notate formal logic – important as a pre-cursor of computing – in a tradition that holds

the origins of the subject of semiotics.

2.1.1.1 Mathematical Notations

One of the few writers on the subject of notation is Florian Cajori (1929), who devoted two volumes to reporting their history. Cajori was surprised to find only one previous book on the subject, and suggested the reason for lack of studies might lie in social context: that the mathematician could not depend on commercial enterprises to exert a "compelling influence" towards uniformity, as they had in science and technology.[1] This historical work is a source of quotations that are reminiscent of some present day concerns and complaints which are reported later in the chapter. In the following extracts, we hear the voices of those early mathematicians who expressed their own aims and principles in constructing notations. All were pioneers in logic and many had experimented with graphical notations.

Cajori draws our attention to Leibniz,[2] who in 1677 advocated the search for a universal calculus of logic: a medium of notation to act as a guide for solving problems, just as the laws of arithmetic support calculation and line drawings aid the work of geometry. Leibniz' vision was not realised until the twentieth century.

In the nineteenth century, notation for logic became an important issue. Augustus De Morgan was concerned about the way that notation had developed without proper consideration, and observed that resemblance, analogy and abbreviation were involved. He preferred simple pictorial notation where possible, and proposed some principles for its design[3]: it should preserve familiar associations, but be free from unnecessary distinctions and ambiguity, so that difference in

---

[1] "The mathematician cannot depend on immense commercial enterprises involving large capital to exert a compelling influence such as brought about the creation and adoption of a world system of electric and magnetic units." (Cajori 1929 p348 §749)

[2] "The true method should furnish us with an Ariadne's thread, that is to say, with a certain sensible and palpable medium, which will guide the mind as do the lines drawn in geometry and the formulas for operations, which are laid down for the learner of arithmetic." (Cajori 1929 p283) [Philosophische Schriften von Leibniz, VII; C.I. Gerhardt, Berlin 1890 p22]

[3] "Distinctions must be such only as are necessary, and they must be sufficient...The simplicity of notative distinctions must bear some proportion to that of the real differences they are meant to represent... Pictorial or descriptive notation is preferable to any other, when it can be obtained by simple symbols... Legitimate associations which have become permanent must not be destroyed, even to gain an advantage..." (Cajori 1929 p327 §713)

symbols would match difference in referents. Alexander Macfarlane (1879)[4], however, criticized the notations (of Boole and De Morgan) for a lack of computational rigour, and blamed the failings of formal logic on its dependence on pictorial notation unsupported by any investigation of the nature of pictorial symbols and laws of manipulation.

At the end of the century, Gottlob Frege originated the earliest formulation of Predicate Calculus, inventing his own pictorial notations, which were entirely individual and unfamiliar. He was aware of the dangers of his "strange-looking formulas". Cajori comments that early neglect of this work had been attributed to its "repulsive symbolism"[5].

Cajori's conclusion was that the problem of creating efficient and uniform notation was serious for mathematicians. He regarded symbolic logic as the major approach to a uniform and universal language in mathematics, but noted that workers in the field had tended to be individualistic:-

> "A question – No topic which we have discussed approaches closer to the problem of a uniform and universal language in mathematics than does the topic of symbolic logic. The problem of efficient and uniform notations is perhaps the most serious one facing the mathematical public. No group of workers has been more active in the endeavour to find a solution of that problem than those who have busied themselves with symbolic logic – Leibniz, Lambert, De Morgan, Boole, C.S. Peirce, Schroeder, Peano, E.H. Moore, Whitehead, Russell. Excepting Leibniz their mode of procedure has been in the main individualistic. Each proposed a list of symbols, with the hope, no doubt, that mathematicians in general would adopt them. That expectation has not been realized. What other mode of procedure is open for the attainment of the end which all desire?" (Cajori 1929 p314 §699)

Since Cajori's day, there have been changes: Commercial pressures have arisen for improvement and standardization of notations – in the software industry. Owing to computers, much obscure notation in mathematical logic has emerged into the industrial daylight. Mathematics itself has gained two candidates for a universal foundation, in the shape of Set Theory and more recently Category Theory, but their notation is far from standard. Yet much has remained the same to the

---

[4][commenting on George Boole (Laws of Thought 1854) and De Morgan (Formal Logic)]: "The reason why Formal Logic has so long been unable to cope with the subtlety of nature is that too much attention has been given to pictorial notations. Formal Logic cannot be developed [in these crudely expressed notations] because the nature of the symbols has not been investigated, and laws of manipulation derived from their general properties." [Alexander Macfarlane, Principles of the Algebra of Logic, Edinburgh 1879 p32] (Cajori 1929 p291)

[5]Frege admits: "Even the first impression must frighten people away: unknown signs, pages of nothing but strange-looking formulas. It is for that reason that I turned at times toward other subjects." [Fundamental Laws of Arithmetic; Monist XXV; Chicago 1915; p491] (Cajori 1929 p295)

present day. Theoretical work on logic (for computation) is a source of notational innovation as it continues to push at the boundaries of the subject, while those practitioners of software development who find most need for notation still largely approach the problem as if they were the first to do so – as we shall see below (§2.2.2).

2.1.1.2 Diagrams, Logical Reasoning and Semiotics

Diagrams have a long history in mathematics. In ancient Greece, Reviel Netz informs us, the word Διαγραμμα means 'proposition' or 'proof' (Netz 1996); the drawing of diagrams was the central practice in mathematical reasoning. The greeks, he considers, did not develop *symbolic* expression; rather their formulaic language was based on an oral tradition, of arguing in the context of a diagram. The diagram had the role of an "inter-subjective object" supporting communication: "The one fixed, solid object in Greek Mathematics is not the word, but the picture."

Closer to our own era, but long before logic was codified, Leonard Euler (1707-1783) developed graphs and circle diagrams as aids to reasoning. The better known Venn diagrams extend Euler's circles into a more expressive form – as recently analysed by Shin Sun-Joo (1994, 1996) and in (Hammer & Danner 1996).

It was not until late in the nineteenth century that full expression of first-order logic (FOL) became possible, through the work not only of Frege, but of an important pioneer of mathematical logic, the philosopher Charles Sanders Peirce[6]. Unlike the now established *predicate calculus* notation in mathematics, both are pictorial.

> Peirce superimposes a hierarchy of nested "contexts", denoted by closed boundaries, on a relational network whose nodes denote individuals (Hartshorne & Weiss 1933 vol.4 book 2; Roberts 1973). The regions in between the nested curves change between negative and positive each time a boundary is crossed, encoding the alternate existential and universal quantification of the enclosed individuals.
>
> Frege's *conceptual notation* (Frege 1972) displays a binary tree of implications, with variables bound to edges as universal quantifiers.

Peirce's Existential Graphs are the subject of a book by Roberts (1973), and papers on Peirce's logic can be found in (Houser *et al.* 1997); also see (Hammer 1996). Annual Peirce workshops

---

[6]pronounced *purse*, according to J.F. Sowa.

have been held in association with the International Conference on Conceptual Structures[7] (Ellis *et al.* 1992, Levinson *et al.* 1993, Ellis *et al.* 1994).

Peirce was a prolific innovator. He also developed a calculus of binary relations, recently studied and extended by Vaughan Pratt (1993) in connection with the logic of concurrency. We find generally that such systems continue to be the basis for many modern notative methods in mathematics and computing, perhaps because computer technology has grown out of the mathematical progress in understanding of logic and calculation. Nevertheless, experts in these fields often prefer formulae to diagrams.[8]

Peirce is regarded as the originator of the subject of Semiotics: the study of sign systems generally. He took a specific interest in the practical problems of mathematical notation (see Hartshorne & Weiss 1933; Hardwick 1977; Peirce 1984). Despite this, most writings on semiotics arise in the Humanities, and take an informal approach that avoids technical detail (e.g. Saussure 1916; Barthes 1967). Semiotic concepts have no agreed formal definitions or mathematical models,[9] and Peirce's work has not been followed up with a full treatment of complex systems of present day mathematical language.

2.1.2 Recent Studies

Although apparently so little current work is expressly dedicated to the topic of notation, there is an increasing interest in its different aspects, and especially in cognitive studies. Here a variety of research directions are reported.

2.1.2.1 Laws of Form

In a similar spirit to Peirce's explorations, an innovative approach to propositional logic notation is adopted by George Spencer Brown (1969). Stemming from his work, some researchers have espoused a 'minimalist' philosophy of mathematical notation, known as "boundary mathematics"

---

[7]The proceedings of these workshops are electronically available. The author is indebted to J.F. Sowa (by electronic mail) for supplying this information.

[8]Evidence for the latter can be found in any advanced text on these subjects. Some writers are explicit about this, e.g. Hoare (1986).

[9]– if we exclude the approach via systems theory, which is not relevant to this thesis. Note, however, that recently published work by Goguen (1997) remedies this; see (§8.2.1) .

(Bricken & Gullischen 1989, Kauffman 1988, James 1993). Although of tangential interest to us here, these systems have found application in the study of biological semiotics by Francisco Varela (Varela 1975, 1979).

2.1.2.2 Written Mathematics

A few authors have begun to pay attention to how mathematics is written. Edwin Coleman (1990)[10] associates the "certainty and independent veracity" of mathematics with its use of notation, and he advocates the consideration of cognitive requirements and consequences of notation and diagram within mathematical prose, which he feels has never been treated seriously in philosophy. Bagchi & Wells (1994) have analysed prose styles; (Wells 1994) contains detailed proposals for making mathematics text more communicative. These pieces of work do not have much to say about diagramming.

2.1.2.3 Higraphs

Graphical notation has been addressed by David Harel (1988). In his work on diagrams for system development, Harel extends the ideas of Euler and Venn. He points out that Euler's circles, which represented logical propositions, rely on the (more recent) Jordan curve theorem – which states that every simple closed curve in a plane divides it into an inside and an outside[11]. The overlapping circles can convey set-theoretic notions, as applied by Venn. Harel proposes diagrams that he calls *Higraphs* to be used for various descriptive purposes, such as for databases, knowledge representation, and for the behaviour of complex concurrent systems using *Statecharts* (Harel *et al.* 1990).

> Higraphs modify and extend graphs and circles to systems of closed curves connected with multiple links, enabling the Cartesian products of sets to be represented, for instance. In *activity charts* the enclosures represent functions and subfunctions; edges denote dataflow.

---

[10] "In the philosophy of mathematics, discourse in english, the epistemological significance of diagrams, of the difference between speech and writing, and of the difference between Word and Notation are all quite generally dismissed."... ...."The objectivity of mathematics, like its certainty, is largely bound up with the use of [notation and diagrams] ... its veracity is independent of the reader's opinion ... If we can understand better why these effects are necessary and how they work, we can put mathematics in its proper place." (Coleman 1990)

[11] This theorem is more subtle than it seems; it states more than is needed for the very smooth curves with few changes of sign in curvature that are involved in practical notations.

Statecharts are intended to answer the known problem of representing specification and design of large complex reactive systems. Harel acknowledges they are unable to represent easily both set inclusion and set membership, except by means of special edges, since two different kinds of insideness are required.

Harel's motivation lies in the observation that the systems to be represented are complex – being composed of many sets that are related in nontrivial set-theoretic ways. His declared interest is in non-quantitative, structural, set-theoretical and relational information, notated by "topovisual" means in which geometric shapes, locations, distances have no significance. He claims that a few simple diagrammatic topological notions provide an effective means of representing systems. From this perspective he discusses notational principles in Entity Relationship Diagrams, Semantic and Associative Networks and Data-Flow Diagrams, noting that graphs are used extensively in computer science. He judges that hypergraphs are less common because they are hard to draw.

Although he has a particular target, Harel remains one of the few researchers to take a broad look at notational problems in a software context, in order to justify his proposals. We would hope for some more theoretical and empirical evidence. Despite the interpretation of diagrams in set theory, Harel does not otherwise attempt to formalize his notative techniques; this task, however, is taken on by Hammer (1993).

2.1.2.4 Visual Language Research

The recent popularity of so-called "visual programming"[12] and visual language in computer interfaces generally (Chang 1994), has led to increased theoretical and practical interest in construction of graphical notation. Tim Menzies (1995) makes an exploration of this "emerging field". His paper presents two overview frameworks: theoretical and evaluative, covering cognitive approaches and empirical studies of language use. It discusses in detail the kinds of expression, purpose and design of visual programming systems – in which scripting is not required. Such systems are held to require a semantic base, a syntactic base, and a set of basic constructs. Menzies emphasizes a connection between data-flow models and production rules.

---

[12]Whilst "visual language" clearly contrasts with "spoken language", it is hard to see why "visual programming" should be opposed to textual (verbal) programming, which is also conducted visually.

He quotes the results of (Goel 1992) who tests the use of "ill-structured diagrams" in solving poorly structured problems found in early system-design stages. Other empirical researches (Green *et al.* 1991, Moher *et al.* 1993) reach similar conclusions, namely that different diagramming techniques are useful for different stages of the design process. Menzies quotes:-

> "Not only is no single representation best for all kinds of program, no single representation is ... best for all tasks involving the *same* program." (Moher *et al.* 1993)

Menzies (1995) quotes some empirical studies by Kindfield (1992), into use of diagrams in biology, which support the view that:

> "[diagrams] serve as an external storage device that frees working memory, allowing for the performance of additional cognitive tasks during the pause when the problem solver is looking or touching the diagram.

Many of the visual systems that Menzies discusses are not based upon notations in the sense of this thesis. He applies two criteria: (1) A visual programming system uses at least two dimensions to represent its constructs, which must be executable. (2) The specification of the program must be modifiable. "A very useful feature of a visual programming system is direct manipulation".

Menzies discusses ways of analysing visual systems. He describes the work of Shu (1986), who defines a Triangle on three criteria:

> visual extent (the intricacy of the visual modality employed),
> language level (high if more effective abstract instructions are offered)
> scope (generality, absolute limitations on expression abstraction)

Though Menzies finds these useful, experience with students leads him to conclude that it is unwise to compare the scope of systems with a different semantic base, and that language level should be measured using subjects who have already been trained.

The range of structures treated in this thesis is more restricted than "visual language" generally. We draw a distinction between visualization of structure encoded in a computer, to help viewers' comprehension, and graphical notation used *by a person* to express ideas. Notations must control size, layout and complexity of symbols, so that they may be hand-drawn during a design task.

Vinod Goel (1992) is one of the few researchers to introduce an awareness of the design process into the discussion on software notations. In another context of design studies, the author (Godwin *et al.* 1997) has analysed the reasons for varying needs of representation at different stages of

design practice. In this thesis, however, we shall put praxeological[13] concerns to one side.

2.1.3 Visual Reasoning

There is now an active area of research into human ability to reason with diagrams (e.g. Glasgow *et al.* 1995; Hammer 1996; Allwein & Barwise 1996). This has partly been prompted by the software industry's problems with human factors in the design of computer system interfaces, where complex information must be displayed. It is also the result of success in using computers to help visualize information (Barwise and Etchemendy 1996).

2.1.3.1 Diagrammatic Reasoning

Zenon Kulpa, in his extensive survey of diagrammatic knowledge representation and reasoning (Kulpa 1994), regards this as "one of the most rapidly growing areas of research in artificial intelligence," – though surprisingly late on the scene. Kulpa emphasises the distinction between diagrams as *analogical* and text as *propositional* representations. He describes the traditional view:-

> "Mathematics has been generally ruled by an implicit dogma stating that propositional reasoning using logic is the ultimate tool of precise and formal thinking. Many mathematicians tend to use diagrams ... as heuristics to prompt certain trains of inference, but mostly only as informal aids to understanding for uninitiated... some of them explicitly stated that the diagram has no proper place in the proof as such."

Taking a cognitive focus, Kulpa summarizes the advantages of diagrams, based upon papers of Larkin & Simon (1987) and Koedinger (1992) :-

> Locality aids knowledge and problem search
>
> There is less necessity for symbolic labels
>
> Diagrams allow easy realization of perceptual inferences
>
> Certain inferences are already present in a diagram [as *emergent* properties].

We notice that this survey places the study of diagrams fully within the domain of cognitive science. A number of researchers choose the same perspective. Work by Keith Stenning (1994) and others has established a way of evaluating diagrammatic reasoning methods which tries to address cognitive constraints. One aspect of these studies is the analysis of differences between textual language and diagrams, in order to establish why and when diagrams assist reasoning

---

[13] I.e. the theory of practice (of using notations in software design).

(Stenning & Oberlander 1992).

2.1.3.2 Media and Modalities

Stenning & Tobin (1994) define *media* as the physical / perceptual aspects of representation systems, and *modalities* as kinds of interpretation function. Hence text and diagrams are in the graphical medium, though they differ in modality; braille and text are different media but the same modality. Their paper aims to give a general account of the cognitive effects of assigning information to different modalities. With the advent of new media, they regard this as an important practical problem; research will be useful if it helps avoid poor designs or speeds up the process of presenting information.

2.1.3.3 Specificity: Direct Analogy in Diagrams

For Stenning & Oberlander (1992), the crucial feature distinguishing graphical[14] and linguistic representations is *specificity*: graphical representations compel the specifying of certain classes of information. The specificity found in diagrams results from the exploitation of homomorphisms (structure-preserving maps) – which Goodman (1968) placed the at the centre of his theory of graphical semantics. Specificity is also employed in natural language discourse conventions, but is not a feature of logical languages. Visual languages that are based on semantic networks also appear to enforce few specificities. Their report contends that diagrams are easy to process because they limit abstraction – hence their widespread use.

Related research (Lee *et al.* 1991) asks what combination of graphics and language is optimal for particular information processing tasks. The paper finds that many features of natural language discourse can be seen as intermediate between logic and graphics (Klein 1987, Oberlander & Stenning 1990). It notes the view (expressed in Tennant 1986) that pictures and diagrams are little more than expository aids, having no place in fully formal treatments of mathematics and logic – which Barwise & Etchemendy (1990a,b) challenge on the grounds that diagrams are sometimes a major aid to theorem proving. The paper draws attention to the "total mappings of identity" that occur in graphical representations, instead of the abstraction favoured in text. It

---

[14]We note that their theory avoids emphasis on the particularly visual; it applies equally to blind reasoners using embossed diagrams (e.g. tactile Venn diagrams have been used). This contrasts favourably with the imprecise use of the term "visual" that we have found elsewhere.

relates this *specificity* of diagrams to the properties of cognitive images and the components of working memory.

Stenning & Oberlander (1992) have undertaken a study of Euler Circles. Based on the analogy of spatial containment, in reasoning these are distinctive in exploiting constraints on movement. This kind of *continuity* introduces temporal specificity: the "mechanical" constraints on discs in a plane helps navigate around the space of models. They refer to the work of Hinton (1979,1980) who argues that such continuity underlies our ability to solve visualization problems. The later paper (Stenning & Tobin 1994) extends this work on Euler's techniques. By comparing several alternative representation systems for syllogisms in detail, the paper explains that the advantage of Euler's Circles lies in their *lack* of expressiveness. Accordingly, they seek to define a distinction between language and graphics, based on the analogical *directness* of the representation (see Stenning, Neilson & Inder 1993).

In a similar vein, other researchers find that logical approaches to these cognitive concerns are useful. By considering *homomorphic* mappings in representation systems, Gurr (1996) is able to provide a precise definition of the kinds of similarity that occur between the structure of expressions and the domain they represent.

These interesting avenues of research provide important insights into notation usage, that we will come back to in Chapter 4.

2.2 Notational Practice in Software Development

This section collects reports of the many views on notations in the practice of software engineering, in order to find out how the attributes of notations may be related to their circumstances of application. Our interest is in interplay of various factors – the *participants*, their subject *domains*, and the notations in various *styles* that play certain *roles* in their *activities* – and in the problems that occur. We will then be in a position to analyse where research in notation may be able to help.

2.2.1 A Preliminary Classification of Notation Use

In the absence of theory about software notations, our objective here is to classify simply the tasks of software development where notations arise, and correlate these with the kinds of notation and

subject areas. This classification does not claim to be complete or definitive, but it will help in giving shape to the succeeding collection of reports, and in setting the scene for the ensuing investigation.

2.2.1.1 Notation for System Design and Software Development

We do not here concern ourselves with questioning the nature of the software design process, which is traditionally described as a sequence of problem-solving phases.

We start by listing the notational needs associated with each phase. The first phase is the capture of requirements, which may employ *knowledge representation* notations – or a 'soft systems' approach which encourages *rich pictures* for discussion of issues in the physical and social environment (Checkland 1981). Next comes system specification, the activity of logically defining required properties, that may involve formal *specification languages*, or a variety of less formal representations. The body of design and development activities often then rely on *structural diagramming* notations. Where appropriate, "formal methods" of development may be applied; they call for the notating of proofs, refinement of specifications and program transformation. At the detailed end of software construction, *programming languages* can represent data structure and all operations upon data.

Peripheral to these activities, there are two more areas of application. The end-users may need a developed system to incorporate their own notations – e.g. specialist professional notations, or visual schematic notations incidental to an operative interface. Lastly, the organization and management of large software projects may be assisted by operational charts or other notation to express progress and version control.

These uses are summarized in the table:-

Area	Kind of notation	Subjects
Requirements	Knowledge Representation	General
Specification	Formal Description Techniques	System properties and functions
Development	Structural Diagrams	System structure
Formal Methods	Logical Calculi	Refinement and program transformation
Implementation	Programming languages	Data structures and operations
User Interface	Users' graphic notations	Users' specialisms, System context
Project Management	Operational charts	Organizational structure

2.2.1.2 A Classification of Software Engineering Notations

Below is devised a short list of some notations used in software development, this time in an order that approximately reflects the history of growth in the industry. We shall not attempt to review these notations, of which there are hundreds, in specific detail. Also outside our brief lie the specialisms of end-users and, marginally, hardware design notations.

Hardware Design: Logic Circuits, Timing Diagrams

Programming Codes:

Assembly codes

Textual Programming Languages

Visual Programming Languages

4GLs, Visual Basic, ...

Programming Aids:

Decision Tables

Flowcharts (PFCs)

Nassi-Shneidermann Diagrams (NSD)

Structure Aids:

Entity-Relation Diagrams (ERDs)

Dataflow Diagrams (DFDs)

State Transition Diagrams (STDs)

Petri Nets (PNs)

Statecharts (SC)

Methodologies:

SSADM

Jackson Method:

Structure diagrams (JSD)

System Specification

System Implementation

HOS: Control Maps, Dynamics Graphs

MASCOT System Diagrams

Object-Oriented: HOOD, MOON, UML

Databases: Query Languages

Knowledge Representation: Semantic Nets, Conceptual Graphs

Formal specification: Z, VDM; OBJ;...

Concurrent Formalisms: CSP, CCS, LOTOS, ...

This list furnishes us with a simple guide for the following reports.

2.2.2 Activities and their Notations

The comments collected here are chosen to reflect areas of concern within the varied tasks and

situations of software engineering practice. They comprise reports upon requirements for notations and practical assessments of languages employed. The reports are arranged in the historical order in which development activities have become important, as a result of the increasing size and complexity of systems.

2.2.2.1 Languages for Programming

Programming languages are a large, well-established and well understood class of notations adapted to purpose. Jean Sammet (1991) reflects that there is relatively little documented history of the thousand or more that have been created. They satisfy varied "functional needs" such as: specialized application areas, different capabilities of user (novice or expert), interactive uses, compatibility with other systems, and demand for a large or small number of features – but the main cause of this diversity[15] is judged to be "the personal needs and interests of people".

We find here that arguments about appropriateness of different languages tend to focus on semantics rather than syntactic style. General issues of expressiveness ("ontology") have for instance been studied informally by Harland (1984, 1986). It is however not usual for programmers to know about the formal semantics of a language. Even so, textual programming language design has benefitted from extensive theoretical study of syntax and semantics. This will not be detailed here, but is touched upon in the next chapter (§3.1).

2.2.2.2 Visual Programming

Though most programming languages are textual in style, graphical forms of notation such as Flowcharts have long been used as less formal ways of expressing program structure. Nassi-Shneiderman diagrams provide one of the first examples of a graphically supported programming notation (Nassi & Shneiderman 1973). Nowadays the almost universal use of graphical interfaces, makes possible many "Visual Programming Languages" that involve interactive techniques rather than simply notations.

Cook & Masnavi (1988), in considering how to make the behaviour of software more accessible to end users, propose graphical methods in programming as a design appropriate for non-programmers. These are suited to programming User Interfaces, where specification of dialogue

---

[15] "...the subject of language design is often a matter of intense debate; in my judgment the bottom line is still that personal opinion plays a much stronger role than any other factor in language design and development." (Sammet 1991)

in a textual language is "an inconvenient and time-consuming task even for experienced programmers".

Brad Myers (1988) classifies programming systems which use graphical representation, and identifies several difficulties:-

(a) Large programs or data could not be easily displayed or viewed, owing to a lack of abstraction mechanisms.
(b) Absence of formal specification might be remedied by some form of graphical grammar.
(c) There was no evidence of worth, in terms of ease and efficiency of use.
(d) Representations were poor, with graphical code hard to understand and edit.
(e) Automation of layout was needed.

He offers the conclusion that for general-purpose programming by professional programmers, textual languages are more appropriate.

According to Menzies (1995), evaluation of visual programming systems is an open issue. He notes a tendency to claim superiority for visual systems (*superlativist* claims in the sense of Green 1991), but finds that although studies suggest there is some inherent utility in visual expressions, experimental evidence yields numerous contradictory results. He judges that for some of these discrepancies the crucial factor determining the value of a representation is not its superficial appearance, but its relevance to the task at hand. Üsküdarli & Dinesh (1995a) note that there are significant concerns regarding success of visual languages, which by consensus are best suited to special purposes within applications.

2.2.2.3 Diagrams in Structured System Design methods

The process of designing programs has often been informal and individual[16], according to Martin & McClure (1985), whose book is something of a manifesto for structured methods and graphical notation. *Structured design* is intended as a more rigorous approach to software development, which concentrates on describing how a system operates, functions or behaves at various levels of abstraction and detail. As well providing help to programmers, the approach aims to improve accuracy of requirements by greater involvement of users.

The "traditional" structured techniques are associated with the names Constantine, Yourdon, De

---

[16] "There tends, however, to be less formality in programming, perhaps because it is a young discipline full of brilliant people who want to make up there own rules." (ibid. ch9 p110)

Marco, Jackson and Warnier-Orr. They are described (*ibid.* ch.1) as a remedy for unsatisfactory results with early programming languages, especially for large programs. In applying the techniques, computers are to be employed at every stage of development, replacing the craft of programming by code-generating tools, with the goal of automatically verified design. There are four basic principles in a structured approach:-

Abstraction, to simplify general form by omitting detail;
Formality, by a rigorous methodical approach;
Divide-and-Conquer, to treat independent subproblems; and
Hierarchy, to introduce more detail at each level of development.

Development is driven by diagrams, depicting overview systems analysis, program architecture, program detail, data structures, database models and file structures. Principles of good diagramming technique are listed:-

They are easily manipulated on screen;
End users can read and draw them;
They are printable on normal paper, or hand-drawn less elaborately;
They avoid mnemonics and unexplained symbols;
Complex diagrams can be analysed into easy modules;
Overview and detail diagrams are similar in structure.

The book (*ibid.* part III) discusses informally many diagramming techniques and their problems, giving a practical guide to achieving good communication. Rigorous use of diagrams is the proposed way to remedy the poor communication that is seen as a major cause of errors and expense in software design (*ibid.* ch9). Fully involving the end-user is "vitally important", hence notation must begin with "user-friendly sketches that the users can draw and argue about." (*ibid.* ch2), so that the notation fulfils an instructive role: "The users should be taught to think about systems with clear diagrams."

Development proceeds by steadily refining these sketches into rigorous designs in a natural manner with computer assistance, but without using a fundamentally different representation or programming text, until code can be generated. Formality is to be provided in mathematical and automated support for diagramming notations, which are seen as a step towards formalization of the development process. The use of formulae that permit axiomatic verification are "not necessarily user-friendly" and therefore not a part of the methods described.

Although the book emphasizes diagramming, we find that it does not completely avoid text. Labelled tree-structures are shown to be most conveniently notated as indented text, and the proposed Action Diagrams apparently amount to little more than graphical annotation of program text. Despite its stress on rigour, the book gives no formal syntax or semantics for any of the notations it describes.

2.2.2.4 Requirements Specification Notations

The need to introduce rigour early in the life-cycle and reduce the errors in defining end-users' requirements has led to an interest in specification languages. The argument in this area has been about the need to use formal notation rather than natural language. Gehani (1985) notes the inadequacies of informal specifications of systems, which "while easy to read, tend to be ambiguous, incomplete, imprecise and overspecific."

Zave & Yeh (1985) describe the specification document as the major channel of communication for development. It "synthesises a collective understanding" of the problem to be solved, forming the basis of a contract. It should therefore be: "precise, unambiguous, internally consistent, sufficiently complete, ... not over constrained", in language that is understandable and modifiable with support of integrated tools for synthesis and analysis, that may assist formal manipulation for verification, and testing for acceptance. Jones (1990 p46) emphasizes that specification expresses what the system is intended to achieve: maintaining relationships while obeying constraints.

Balzer & Goldman (1985) require Specification Languages to provide means to represent a dynamic model of the system's environment, that expresses data uniformly as relations among objects, independent of their representation, and with facility for descriptive reference. Techniques are to allow specifying by extending analogous concepts. They do not consider syntax needed to achieve this.

Tse & Pong (1991) describe the features of specification in natural, formal and graphical languages that are desired by other authors. They write that notations should enable one to "analyse and manipulate a model abstracted from the real world," in order to produce a solution. This abstraction must be easily understood by all concerned; the use of familiar language would also help reduce staff or management resistance. They admit that natural language text improves

user understanding, as it gives better persuasive power and freedom of expression in the initial phases where there is uncertainty; however ambiguities are caused, and there are unsolved problems in manipulating it. Thus in engineering, diagrams and mathematics are used because they are more easily manipulated than verbal descriptions; and diagrams may be converted to equations. The need to transform one representation into another prompts them to prefer formal or mathematical text, which is to be explained through natural language. Formal correspondence between the various syntaxes of both formal and informal versions must be maintained.

They see *graphical* language as more comprehensible because its two dimensions help express hierarchy and parallelism, and graphics can be read selectively rather than in sequence. Provided there are not too many symbols, the reader can focus on overall structure before inspecting details. They therefore propose hybrid notations that combine graphical languages best for overviews, with formal text preferred for detailed description.

They claim that complexity is the main barrier to understanding, and propose to overcome this with the following structural norms, which aim to improve conceptual clarity:-

> Separation of concerns, essential versus physical;
> multi-level abstraction, hierarchical 'top-down' visualisation;
> structuring requirements into parts that are easy to modify;
> language to allow a natural and logically verifiable process of refining systems into subsystems;
> self-contained Subsystems with minimal interfaces between them.

In order to support the necessarily different models, depending on environment, emphasis and stage of development, several very different notations are usually required. Hence they propose the ability to transform between styles/ notations with respect to their mathematical semantics, but without exposing untrained users to unusual symbols and jargon. These conclusions are echoed by the views of Cohen *et al.* (1989), who write:-

> "Debates are held on such topics as graphics versus text, readability, ease of use, ease of learning and compatibility with existing tools. However little consideration seems to be given to the roles of such languages in the design process or to the relationships among the descriptions of systems expressed in them."

Guttag & Horning (1985) report on their own experience of creating a specification, which was facilitated by inventing notational shorthands. As designers they found this compactness in notation greatly helpful, but it was a hindrance for uninitiated readers, who were therefore offered

an easier yet semantically equivalent style in their paper. They infer a need to "maintain semantically consistent, but notationally distinct versions of the same specification." Also Zave (1985) highlights the need for different styles of notation to suit non-technical participants, and looks forward to tools which derive simpler reports and diagrams automatically.

These various reports reflect a desire for rigour to cope with complexity, but they focus upon system design requirements rather than syntactic requirements.

2.2.2.5 Formal Methods

In opposition to the popular emphasis on diagramming, Hoare (1986) proposes a professional engineering approach based on explicit mathematical laws. He does not approve of the use of pictures such as flow charts, because they "inhibit the use of mathematics in programming". Hoare views a specification as an abstract program, that will be refined formally into a concrete implementation. This abstract program (which may well not be executable) delimits the set of alternative acceptable systems, and requires "the full range of concepts and notations of mathematics". We infer that specifying is a more complex task than describing behaviour of a single system.

Cohen *et al.* (1989) likewise observe the need for "sound scientifically based formal methods" to transform software development from its 'craft' status into a true engineering discipline. For this purpose, they propose that languages must "possess the primitives and constructors necessary for the expression of complex models, together with semantic definitions, calculi and proof rules which permit the properties of such models to be deduced."

Goguen (1985) prescribes that a specification language have a formal definition in terms of "some underlying logical language having a precise mathematical semantics and a set of inference rules which is consistent and complete", if it is to serve in formal verification.

Notation for methods based on mathematical logic is rarely graphical; we find few diagrams used in standard books on formal development (e.g. Jones 1990; Dijkstra 1990; Potter *et al.* 1991) where they are mostly confined to informal illustration. Many formal specification notations make little use of diagrammatic features, with the minor exception of schema-boxes in *Z* specification. Formal diagrams have found more of a role in describing concurrent and communicating systems: for example Petri Nets, which rely on graph theoretic formalisms.

As with the previous, these writers express the need for underlying logical formulation, but only formalize the notations in a limited way, that disregards its graphical characteristics.

2.2.2.6 Knowledge Representation

Formalizing the requirements specification may be seen as a problem of Knowledge Representation. This topic is studied in the context of databases with inference mechanisms, and construction of query languages (Brodie *et al.* 1984). They identify three approaches: semantic networks, production systems and logical schemes. Mylopoulos & Levesque (1984) report that methods of representation suffer the drawback of lack of formal semantics and standard terminology.

2.2.2.7 Conceptual Graphs

We have noted above (§2.1.1) some of the early attempts to find a graphical notation that can serve as a logical language. These techniques have in recent years been further investigated and extended, by John F. Sowa and others – with the aim of allowing requirements to be formalized in logical propositions and expressed in diagrams. Sowa (1984) uses ideas of C.S. Peirce and semantic nets (Sowa 1991) to design a pictorial language which he claims to be formalizable. *Conceptual Graphs* are a typed (sorted) version of Peirce's existential graphs; they provide a basis for organization of knowledge, formal and informal logical inference and computation. He draws on research in cognitive psychology to support his method, and invokes Hintikka (1969) to extend Peirce's techniques to modern logics. Sowa recognizes the need for lambda-abstraction in his notation, but does not give a pictorial expression for this.

Sowa provides a clear indication of an equivalent textual form for his Graphs. We do not find a formal treatment of his notations, though elsewhere he alludes to graph grammars as a possible formalization (Sowa 1979). Sowa encodes negation as a property, rather than a modifier, of a sentence; although this works in practice, it seems less elegant than Peirce's version.

Loucopoulos & Champion (1990) apply Conceptual Graphs to the early requirements analysis, which must tolerate informality of expression. They find that the Graphs provide an unifying representation formalism for the user's concepts, and avoid confusion with the semantics of the development method. Application of Conceptual Structures has also been the subject of several conferences (Tepfenhart *et al.* 1994; Ellis *et al.* 1995).

The system of Conceptual Graphs is a rare example of how formulae in predicate logic can be diagrammed. It is a complex notation, however, with hierarchical enclosure of sub-graphs used for modal statements and negation. Interestingly, the graphs have topology not unlike Harel's, despite a very different semantics. Whereas Harel uses them for a variety of specific semantic purposes, Conceptual Graphs have uniform underlying logic, in which the depth of nesting shows the logical complexity of the expression.

Most of the authors quoted above have given consideration to the suitability of notation for a particular type of task, without proposing any integrated approach to notational problems – other than increased rigour and more use of graphics.

2.2.3 Reviews of Notations

In addition to these general views, there are a few studies of the adequacy of particular notations.

Alan Davis (1988) compares specification techniques and commends the simplicity, applicability and elegance of some of Harel's notational techniques in Statecharts, which are expressive and compact. Even so, he concludes that the more sophisticated rigorous graphical notations such as Petri nets, PAISley and Statecharts are much more difficult to comprehend for non-professionals. Harel (1988) himself expresses enthusiasm for the future of visualization, predicting that daily technical language will be inherently diagrammatic, perhaps also three-dimensional and animated, to encourage both old and new visual modes of thinking when tackling systems of ever-increasing complexity.

Tse & Pong review (1991) several languages in computer-aided software development (PSL, SADT, EDDA, SAMM, HOS, RSL), which they mostly find to satisfy the list of structural criteria quoted above (§2.2.2). Many languages resulted from studies in formalism, however, which they saw as causing a psychological barrier to end-users. They felt that the main user-interface should avoid formalism, via an interface with popular tools such as structured methods.

Hull *et al.* (1991) compare four real-time applications development methods (MOON, HOOD, JSD, MASCOT) that feature graphical notation. Their criteria for evaluation are:

ease of use and understanding; use of diagrams;

ability to express requirements and constraints;

ability to express concepts of the subject domain;

simplicity and compactness of syntax;

structural features: modularity, hierarchy, viewpoints;

formality: unambiguous, consistent, implementable;

support for methods: phases of development.

None of the four are found to satisfy all criteria; they suffer from lack of abstraction mechanisms, inappropriate styles, lack of rigour, over-complexity, and inability to express requirements and constraints. At lower levels the authors find diagrams cumbersome for sequential actions – text was preferred. Otherwise, effective notational techniques are evident between the four: good abstraction mechanisms, support for concurrency, rigour, hiding of detail, and appropriate semantic range.

These reviews, though helpful as a guide to concerns, are based on subjective assessments. We might ask what kind of theoretical study could support a more objective test of notation adequacy.

2.2.4 Formalization of Notations

We have seen that a desire for rigour in notation is common to many of these reports. This section ends with some further collected views on the issue.

Martin & McClure (1985 p17) remark that formality enables the study of programs as mathematical objects, the clear communication of ideas and instructions unambiguously in a computable form, and a way to focus creativity, thereby blending the craft and engineering aspects of programming. But this formalization is not to be expressed by formulae; methods (such as HOS) are preferred which constrain the design process to be correct.

A detailed case for formalization of specification and design language is given by Cohen *et al.* (1989 p99), in criticism of common informal approaches which do not meet the objective of re-usability. They propose that languages be formalized in order to express and deduce properties of complex models; they should have a tractable syntax and well-defined semantics. *Ad hoc* notational extensions are rejected in favour of providing soundly based semantic and syntactic flexibility by means of embedded extension mechanisms (capacity for definitions). In the work cited they also observe that formalization must also relate different notations needed for different

aspects of behaviour.

Tse & Pong (1991) claim that notation formalism helps to reduce misunderstanding between designers and enables automated consistency and completeness checking. A unified mathematical framework must be present in a specification language: verification relies on relating the development to such a theory. They note that frequently the theoretical background adequate for rigour is absent.

Spivey (1988 p7) discusses the applied mathematician's disregard for formal semantics, and gives practical reasons for taking a different attitude in the case of the specification language *Z*. He finds that explicit formalization is necessary to explain *Z*'s unusual modular structure, and to enable reasoning about specifications. It also provides a means of comparing specification techniques and language constructs.

2.2.4.1 Need for explicit formality

We note a variation in these comments regarding the strength of the term 'formal'. Formality in a weak sense may amount only to an implicit enforcement of conformity between representations, by computer-aided (CASE) tools. Explicit formalization is a stronger notion, relying on mathematical models of linguistic and graphic syntax, related to semantic models for each subject domain – which is rarely available. Only for some programming codes and formal methods notations are strongly formal. The views quoted suggest an awareness of need for weak formality, but only in a few cases do we find requests for the stronger, explicit formality.

2.3 Analysis and Discussion

From the comments in these reports, we wish to assess in what ways notations are succeeding or failing to fulfil the roles allotted to them, in order to identify where research is needed. To help in the assessment, this section analyses and summarizes the issues and difficulties that have been noted, and considers their causes. The discussion concludes with an evaluation of the state of affairs and some suggestions for a programme of work in this area. .

2.3.1 Analysis of Notational Issues

The reports of the previous section are mostly *situated* in the professional context of software

engineering, and cannot be regarded as grounded in empirical objectivity. Because of this, an analysis of the opinions presented tends to take on an ethnographic flavour. Rather than taking at face value the views found, we hope to bring out the issues that require deeper investigation.

Considering the force of the views surveyed, the difficulties and their causes, we arrive at the position that formalization is a necessary step in improving the design of notations.

2.3.1.1 The Need for Visible Formality

The first issue to notice is that of increasing formality in the use of language and notation. If we follow the historical sequence from elementary mathematical logic to computer-aided software engineering, we find a trend towards greater rigour and formality, associated with widespread use, mechanisation and increasing complexity of problems tackled. This has brought about a conflicting need to communicate widely and technically. Standard diagrams are desired for communication with both end-users and developers, while the less accessible notations of mathematical formulae are selected to provide a source of rigour and accuracy, through deductive method.

Though both coded programs and supporting proofs are therefore hidden from general view, the unambiguous specifications of requirements and description of developed solutions must somehow be made appropriately visible to all participants. Thus Cohen *et al.* (1989) stress the importance of differing roles of languages in the design process and the need to understand relationships among the varied descriptions of systems. In the views collated below, we see that mathematical and computational support are called for in order to manage these relationships.

2.3.1.2 Tabulated Views

We can analyse many of the views into claims, wishes, problems and fears about language needed for design representations. In this analysis, comments on language and notation are extracted from the reported views and organized into four tables. The specific role taken by the representing language is mostly filtered out, in order to highlight general issues. To begin with, we find references to the advantages and disadvantages of natural language (in the form of text).

For natural language:-

Claims	Problems
better persuasive power and freedom of expression	incomplete, imprecise, overspecific ambiguities are caused (twice) less easily manipulated

Replacement of informal writing by formal textual language carries a different set of concerns. It raises a worry about communications, and a desire for translation.

For text / formula:-

Claims	Wishes	Fears
can transform one representation into another	explained through formally corresponding natural language	formulae in axiomatic verification may not be friendly. exposing untrained users to unusual symbols and jargon

There are a number of general views about notation, which point to the conflict of needs: for simple expression of technically complex subjects. Here the worry is about precision and comprehension, with mathematics and logic needed for support.

General concerns:-

Wishes	Problems
must have primitives and constructors to express complex models separates conceptual from concrete precise, unambiguous, internally consistent, sufficiently complete, not over constrained understandable and modifiable	complexity is the main barrier to understanding Specifying requires the full range of concepts and notations of mathematics those resulting from formalisms, cause a psychological barrier to end-users
integrated tools for synthesis and analysis allow logical refinement into submodules automatic verification; formal manipulation for verification and testing	
with semantics, calculi and proof rules; precise mathematical semantics, inference rules ability to transform between styles/ notations; maintain notationally distinct versions of a specification	knowledge representations lack formal semantics and standard terminology programming notation diversity results mainly from personal needs and interests
a hybrid of graphical and formal notation abstraction, hierarchy, modularity	

The latter two wishes are for an integration of diagrams and formulae, with structuring facilities that are found in natural language narrative. Diagrams on their own only partially solve the concerns; contradictory views are found on their efficacy, reflecting varied success over the difficulty of dealing with complexity.

For diagrams:-

Claims	Wishes	Problems
a remedy for poor communication more comprehensible users can draw and argue about	users learn to think about systems	no evidence of ease and efficiency in use hard to understand over-complexity inappropriate styles inability to express requirements
can be read selectively user can focus on overall structure 2D helps express hierarchy and parallelism	automatic layout abstraction, hide detail modular, hierarchical.	Large programs or data not easily displayed hard to edit lack of abstraction mechanisms (twice) cumbersome for sequential actions
may be converted to equations	rigorous, methodical formality provided in mathematical and automated support generate code from diagrams	rigorous diagrams harder to comprehend for non-professionals lack rigour use of pictures inhibits the use of mathematics

In summary, although graphical notations are held out as an aid to clear thinking and a better means of communication, the concern about rigour brings out a need for formalization. The rigorous diagramming methods adopted in some CASE tools may fail to be communicative, however, and are not as compact as the formulaic text which skilled users still value -- even if rules for formal reasoning are not known explicitly.

Researchers in Visual Programming for example now consider it unhelpful to make a blanket comparison as to which is best, text or graphics, since the two modes are suited to differing purposes. We also find no clear justification for a theoretical separation between modes, given the *visual* quality of text, and the fact that they are in practice mixed together. The evidence noted in (§2.1.2) suggests rather that we view notation comprehension as a collaboration between two specialized cognitive abilities – distinguished as linguistic and spatial (amongst others).

2.3.1.3 Causes of Difficulty

The above analysis brings to light several points of difficulty. It is seen that notations must cope with a wide range of kinds of system, and express different aspects of behaviour. Because they must be used in different ways by different participants to serve different purposes, we find many conflicting features and styles of expression. Design of notation is difficult, and requires development of both mathematical logic and pictorial metaphors. In order to cope with the complexity of large software systems, notations become abstract and hence hard to reason with. Systems can be described using mathematical or computational language, but this kind of

formality reaches only a small audience.

What can be done? In order to reconcile the conflict between differing needs of participants in software development, computational support will be needed for all styles of notation (formulae, text, diagram). An explicit means of defining notational *semiosis* could offer users a more adaptable access to varieties of notation which better suit their cognitive abilities and prior knowledge. If we wish to have an objective test of notation adequacy, then an analysis of cognition is surely important, though it is not a prerequisite for formalization of functional structure.

The difficulties noted go against the view that graphical notations should only be used informally – for explanation to users or for illustration in the development task. If instead we could supply a logical basis and computer support for notations, we could make possible a more flexible style of expression in standard notations, thus aiding both human reasoning and mechanical calculation. By formalizing, diagrams could be given precise meaning, to enable accurate communication with users, but without prohibiting the use of informal variations and annotations. Tasks which insist upon rigour may be made easier to grasp by the use of computer-aided manipulation of diagrams, made possible by formalization.

2.3.2 Conclusions

To round off the discussion, the weaknesses identified in the survey are summarized, and seen to result from a lack of support for notation design. It is argued that changing circumstances necessitate continuous development and invention of notations. In view of these considerations, possible directions for useful research are suggested.

2.3.2.1 Weaknesses and Concerns

The survey reveals that there is a lack of serious studies of notation, and little general theory which might support empirical investigation. Scientific observation of usage and study of cognitive aspects has lagged behind practical work on language design. Recognition of the need for rigorous semantic definitions has been belated, when it is acknowledged at all.

There are reports of practical assessments of particular languages, and of requirements for notations fulfilling some role in the software lifecycle, but these references make little separation between notation and method. They suggest that the main difficulty with notation lies in

adaptation to the preferred styles of users or to the range of problems to which they are applied. Expressive principles uniform across different notations have not been established, which makes it hard to extend syntax and semantics when changes are called for.

From the reports received we learn that users and designers of notation wish to avoid ambiguity and vagueness of meaning. As well as precision, there are practical concerns about training needs, effectiveness and compatibility with methods. Though there is a consensus on need for simplicity, accessibility, familiarity and expressiveness, there are no agreed principles for evaluating designs in these terms.

2.3.2.2 The Need for Notation Design

Why is the designing of notation a cause for concern? What is its importance?

We observe that notations arise to fill people's need to express and share ideas about technical problems; they support storage, communication and re-use of ideas, and are important as an aid to formal and informal reasoning and calculation. Within software engineering, notations provide means of thinking about a wide class of problems. Despite this, choices made in notation design have not been justified theoretically or empirically. The pattern has been for individual practitioners or groups of researchers to propose notational innovations and build languages based on them in an *ad hoc* manner. The chosen techniques have been assessed informally by popularity and usage of the languages.

The pertinent question is not to decide between formulae and graphics, mathematics and intuition; it is how best to combine effective pictorial and linguistic metaphors within a formal basis – even if a notation is to be used informally. In computing there is no fully established "universal calculus" (as desired by Leibniz for mathematics) but there may be some hope for a universal thread connecting the known notational techniques.

Notations are not only a means of talking to other members of the profession. Unlike the situation in mathematics, software engineering notation must sometimes be effective in communicating with non-professionals. In the interests of safety and effectiveness, we would wish to design notations that can express complex systems accurately in forms helpful to the understanding of the various participants in computer system development. It would surely be unwise to develop a software system so complex that it was humanly impossible to describe. New notation can be

powerful when expressing what was formerly too difficult. Notation design is then not just about convenience, it is about understanding the subject domain.

2.3.2.3 Pressure for Change

Why should there be a need for design at this time? We can point to several factors. Firstly, visual languages are under development in many areas of application, aimed at many different end-users. The number of alternative notations and variants which exist, the lack of standards and tools, the arguments about comparative merits, all indicate that notations will continue to change and develop with increasing use. There is need for flexibility and adaptability in work with notations.

Secondly, from the cognitive approaches in (§2.1.2) it appears that for a diagram to be effective, its visual structure must be linked by analogy to aspects of its subject domain. It follows that new understandings of subject areas, growing out of improved metaphors or better logical characterizations, bring about semantic change that will require improved notations.

The third factor is that new theories of software behaviour are certain to arise. A semantics for concurrent interaction of communicating systems is still a matter of debate. Computer systems are increasingly complex and frameworks for rigorous project development methods are still being sought. We can therefore expect invention of notations of all kinds, in order to cope with the resulting difficulties of expression.

A final reason is that the developing skills and experience of the many users of notation may also have an effect on syntax preferences, even when semantics remains relatively stable.

2.3.2.4 Some Research Avenues

There are several directions that research could take in order to improve matters. Our understanding of the problems would be greatly improved by:-

- a comprehensive examination of many software engineering notations
- an investigation of cognitive properties of diagrams, text and formulae
- an empirical study of actual usage of notations in software development

By such means, from existing examples, we might learn which attributes of notation contribute to fitness and success, and in this way establish design principles. Each of these means, however,

requires a clearer theoretical foundation than has yet been developed. Therefore we might make:-

- an attempt to improve upon current methods of describing notation structure
- an attempt to improve upon current tools to process notation

The difficulty here is that most of the relevant work in these directions is being carried out in parallel with research for this thesis. In the next chapter we shall review the diversity of approaches to techniques and tools. In any case, the principal research need that has been identified in this chapter is for:-

- mathematical and computational support for notation design and processing.

The detailed research agenda chosen for this thesis will be presented at the end of the next chapter.

2.3.2.5 Summary

We have found in this chapter that the use of notations is bound up with logic and reasoning and has a long history, though most theoretical interest in how they work is very recent. Hardware design has perhaps more of a technical tradition of notation, crafted from practice with earlier electronic devices. In the subsequent historical growth of software development practice, we have identified many notational concerns:-

- The primary requirement for programming 'languages' has been addressed in a piecemeal fashion, though the problems engendered have lead to an awareness of the need to understand linguistic structure and to provide formal semantics.
- Systems analysis has been served in the main by informal diagramming techniques, based on no clear formulation of the development process, and lacking any notational theory.
- Requirements analysis suffers lack of support for precise means to communicate with users.
- Formal specification and development techniques have adopted much of the style of mathematical formulae, with little concern for ease of comprehension.
- Pictorial expression of programs has recently become popular, stimulating considerable interest in the usefulness of diagrams, but able to claim no definite empirical advantage.
- End-users of systems have their own notational needs, which have not been systematically studied.

An argument has been made for formalizing the description of notation in all its different uses, based on the need for computer-assistance, and the specific need for precision in many stages of software development. It has been proposed that the activity of notation design must be supported if the practice of software development is to be flexible enough to keep pace with changes in power and areas of application of its technologies.

We are led to infer that an uniform theory of notation structure is needed before it will be possible to offer versatile support. Without appropriate theory and science, no clear design principles can be relied upon to recognize and resolve problems. Only theory will allow us to frame the right questions about choice of modality and language for the many aspects of the software development process.

Chapter 3

Review of Notation Support in Software Engineering

Abstract

Here we find a review of techniques for definition of syntax and tools for developing notation editors. A compilation of reported problems and solutions leads to a statement of the research agenda for the thesis.

First the review looks at theoretical techniques for formalizing syntax and semantics of languages, using mathematical or computational approaches, that could also apply to notational description. It then looks at formalisms that attempt to specify notations and visual language. The review shows that only during the later phase of this research have reasonably general methods of definition become available. These methods are based on a variety of theoretical principles, which are hard to compare or combine; the main trends are to apply specification logics, spatial theories, or graph grammars. Semantics is treated operationally, though algebraic methods are used to explain visual analogy. Grammar-based approaches are common, but graphs cause difficulties that are not found in string or term rewriting, and grammars must be augmented with spatial constraints. Some attempts are reported to place grammars within a general hierarchy according to expressive power.

The review reports on the capabilities and limitations of the available aids for notation-processing. Reports of generic editors show that the task of developing an editor for a notation can be speeded by methods of graphical and syntactic specification such as graph-grammars. Theory has not, however, kept up with practice, and it is not clear how a designer of notations could reason about the syntax that is built up. Only the simpler grammars can easily be parsed, and it is not known how widely these can apply.

These methods leave us with a jigsaw puzzle, whose pieces can yield only a patchy picture of several related scenes. The proposed way forward is to treat a notation as a sign-system; this function, rather than pictorial appearance or semantics, should determine its formal structure. The aim of this work is then to establish an uniform descriptive theory that can offer practical help with design and processing of notation.

Chapter 3.

Review of Notation Support in Software Engineering

This chapter reviews research on computer-assistance for the uses of graphical notation in system specification and development. The review selects contributions that may resolve the problems observed in the previous chapter (§2.3), where we have seen that notation has very little in the way of design science. We therefore hope to find standard techniques for formal description and tools that can cope with the complexity and diversity of notational needs, easily allowing combination of diagrammatic, formulaic and textual expression.

The review first looks at theoretical techniques, from mathematical and computing fields, whose aim is to formalize language and notations. It then considers what is required of tools that aid in processing notations, and reports on the capabilities and limitations of those that are available. Reports on current research directions strengthen the argument in favour of developing a uniform basis for notational design. The consequent discussion of potential and problems in the material reviewed leads to a statement of the research agenda for the thesis.

3.1 Formal Techniques for Describing Notations

The purpose of this section is to review all the important published approaches to the problem of defining the structure of graphical notations.

3.1.0.1 Inadequate Methods of Description

The material in the previous chapter indicates that there is no tradition of rigorous description. Instead, one of the more careful approaches is to relate graphical syntax directly to a textual language in an informal way, and rely on a formal syntax for the text – as is done in the MASCOT handbook (1987), using Wirth syntax diagrams and Backus Naur form (BNF). Inevitably the textual syntax does not cover the many connectivity constraints observed in the System Diagrams, since they translate to restrictions on naming of variables in the same syntactic category, that are generally not expressible in BNF. Nor does this method make formal reference to the spatial nature of the notation.

Those authors who adopt formal or mathematical techniques, also mostly refer to syntax of

diagrams entirely informally. For example, Tse & Pong (1989) prescribe a formalization of Data Flow Diagrams via the algebra of Petri nets — which necessitates extending both of the notations they treat. They insist on graphical expression and formal method, with clear use of mathematics for semantics, but they omit to describe how the diagram syntax is to be formally manipulated.

Less formal still, are the presentations usual in standard texts on software development such as (Jackson 1983; Martin & McClure 1985). Diagrams are described directly in terms of the concepts represented, with the aid of natural language and examples.

In practice, for existing notations, we thus find the formality required is often lacking. Without this, tools must employ a variety of programming languages to embody syntax, as we see later (§3.3.2). The infrequency of proper formal techniques has been criticized by other authors. Rekers & Schürr (1995b) find it regrettable that new users can only guess the syntax from provided examples. They believe it would be very beneficial to agree upon a single syntax definition formalism — as long as it is highly expressive, unambiguous, with specifications that are easy to read and develop.

3.1.0.2 Theoretical Remedies

Since the methods of description in textbooks are not found to suffice as a basis for notation processing, we turn to theoretical studies of this problem. Largely the recent theoretical approaches have taken their cue from various means of formally defining how *verbal* languages work. We therefore first take note of the background work on study of grammar and semantics of language generally; the problems and methods of structural linguistic description are considered relevant to an understanding of the similarities and differences between 'natural' language and the more artificial world of notation. These methods are extended to the case of graphical notation with the help of a notion of 'graph', which is intermediate between linguistic and geometric structure. In the next section (§3.2), we find that this kind of work forms the basis of many reported methods for defining notational structure.

3.1.1 Formalizing Language Syntax

We begin with an outline of the standard theory of formal grammars, in preparation for investigating below how theory has been generalized to graphical notations.

3.1.1.1 Formal Textual Languages and Generative Grammars.

In his seminal work on the problems of natural language description, Noam Chomsky studied various mathematical idealizations of language, which feature a strict separation between syntactic and semantic aspects of structure. These formalizations have proved to be of value in their own right, forming the basis of design for artificial languages, notably for programming.

> A *Formal Language* is defined as a set of sequences ('strings') of symbols from a finite set, called its *alphabet*. Its syntax gives a way of specifying which sequences belong to the set, and may be presented by a *generative grammar*.

In generative grammars, expressions are constructed by applying *rewrite-rules* (also known as *productions*). The adequacy of formal grammars is assessed in terms of how large or complex a class of languages can be specified by a certain kind of grammar.

Different kinds of grammar give rise to four major families of languages generated, which form the "Chomsky hierarchy" (van Leeuwen 1990 p109) from type 0 to type 3. Each language L of the most general family (type 0: *recursively enumerable*) can be generated by applying a finite set G(L) of literal substring replacement (rewriting) rules. The pattern of replacements culminating in a given expression is called a *derivation* of the expression (in the grammar G); before the expression can be interpreted, this valid derivation structure must be discovered by a process called *Parsing*, which is thus a first stage to any semantic processing of the language. The importance of the types of grammar is then evident, for they determine the complexity of automatic parsing; type 0 languages are in general impossible to parse, and are therefore too complex for practical languages.

The preferred choice for artificial programming languages is *context-free* grammars (CFG) (van Leeuwen 1990 ch2), which define any type 2 language, and are commonly written in BNF. A CFG grammar consists of rules for rewriting single meta-symbols ('nonterminals', which stand for phrase types, and augment the language's alphabet). The generative process starts with a nonterminal (typically 'S' or '<sentence>') and is completed when the rewritten string contains only symbols from the 'terminal' alphabet. Parsing discovers a hierarchical decomposition of expression structure, a derivation that can be drawn as a labelled oriented tree, whose leaves constitute the sentence. Syntax rules that constrain the *type* of an item, however, often require a language of type 1, and *context-sensitive* grammars are needed (van Leeuwen 1990 p372), which set

conditions on when a symbol may be rewritten.

3.1.1.2 Syntax for Natural and Artificial Language

Although useful, these notions of grammar cannot capture the subtlety of structure found in Natural language, either in getting the 'right' set of sentences, or the 'right' derivation structure. Chomsky (1965) proposed to remedy this by means of transformations on the derivation tree, as a second phase of production. An example from English is the formation of a passive sentence from a corresponding active one. Unfortunately such *Transformational* Grammars can in principle generate any recursively enumerable language, which is not justified empirically (Skousen 1975).

The power of such formal approaches suggests that general syntactic structure is a complex matter. This is apparent in the various interesting formalizations proposed; for instance, *Arc Pair Grammar* of Johnson & Postal (1980), involves an elaborate graphical notation that is backed up by predicate logic. According to Carpenter (1995b) however, no formalism has even come close to providing a universal system in which all and only natural language grammars can be expressed. Nor have grammars for particular languages come close to covering a naturally occurring range of data in a theoretically clean fashion.

Some of this difficulty in defining the 'right' syntax arises from attempting to separate syntax from semantics, or rather from trying to include too many semantic factors. According to Chomsky (1965), "any attempt to delimit the boundary [between syntax and semantics] must be tentative". Hintikka (1979) also disputes whether syntax can be completely independent of semantics. Montague (in Thomason 1974 p210) states that the construction of syntax and semantics must "proceed hand in hand", as in syntax there are too many irrelevant ways to generate sentences.

3.1.1.3 Design of Grammar

Chomsky (1965 p62) advises his readers to seek the simplest theory of grammar which is empirically adequate. Since our requirements, in formalizing notations, are more limited than those of natural language, we should likewise avoid adopting too complex a view of language from its natural examples. Johnson & Postal (1980) maintain that such subtleties are unneeded in mathematical language where form directly relates to its logic. Yet their own notation for predicate

logic demonstrates counter-examples[1] to this statement. Whereas Montague (In Thomason 1974 ch6 and p216) disputed that any important theoretical difference exists between formal and natural languages, he also held that if a language is to avoid ambiguity, or is to fit within a first order framework, then its theory of descriptions should not try to mirror English closely, but be influenced by simplicity.

It has been the practice in programming languages to use grammars which are simple to parse, but retain some syntactic flexibility. This indicates the balance that must be struck between simplicity and flexibility in any kind of notation; it calls for a sufficiently general theory of grammar, but one that still allows efficient and unambiguous interpretation of expressions. There is continuing interest in making it easier for users to engage with the newer languages, which may be helped by a more sophisticated model of grammatical structure than has been customary in formal textual notation.

3.1.2 Formalizing Semantics

We next look briefly at studies of formal semantics of natural and artificial language (following Richard Montague and others) that use symbolic logic and model theory, that are equally available for treating diagram semantics. In the reports, their authors take various approaches to the analysis of meaning. The reason for this variety of views on semantics may relate to the different operations that are required on linguistic expressions – from type-checking to translation, logical deduction and computation. An appreciation of the context of use of expressions is properly the concern of *pragmatics*.

There are evidently two distinct aspects to semantics as it affects notation processing. (1) The formulation of rules of acceptability: whereby an expression may be rejected if in all feasible situations it would appear anomalous, and (2) a formal interpretation of the meaning of expressions: in particular, their *denotation*.

---

[1] It is common to write **a,b < N** although there is no such object as "**a,b**"; it is certainly not the ordered pair **(a,b)**. Johnson & Postal themselves innovate the notation of first order logic by writing **P(x&y)** for **Px & Py** , a familiar device in natural language; but no such object **x&y** logically has the property **P**.

Other examples of a subtle connection between syntax and semantics can be found in differential calculus; the sign **dy/dx** is syntactically but not literally a fraction, and it also hides the functional dependency of **y** on **x**, which easily leads to ambiguity.

In regard to (1) we have observed that in linguistics a clear boundary cannot necessarily be drawn between syntactic rules and semantic rules. For formal languages, there is however a substantial difference between the rule-systems used for grammars and those used to determine logical validity, in proving semantic properties.

In (2) several points of view can be found. Frege distinguished the denotation from the *sense* of an expression, as a collection of instructions (as discussed in Girard *et al.* 1989, ch.1); the *denotation* is a kind of ideal result of these instructions. This is a distinction that Montague found useful (only) in certain artificial languages (Thomason 1974, p217). Galton (1988) suggests that if denotations are themselves notated, formal semantics may be reduced to a case of translation. Joseph Goguen remarks that semantics is a quotient of syntax; i.e. the expressions of a language divide into equivalence classes of those having the same meaning. There is also the *formalist philosophy* that does not acknowledge the need for semantics, but expects meaning to reside entirely in formal rules for manipulating expressions – a view expounded by Hilbert[2] .

Accordingly, we note here some common systems of symbolic logic, followed by denotational methods, and then the semantic grammars that have found favour in computational linguistics, ending with pragmatics in the form of Situation Theory.

3.1.2.1 Symbolic Logic

Logical systems generally can be viewed as ways of formalizing semantics, since they provide formal languages designed to capture the meaning of natural language sentences. Below are summarized those standard systems that are most commonly referred to in the reports in the next section (§3.2): Predicate Logic, Algebra, Higher-Order Logic, Set Theory, and Category Theory.

Predicate Logic allows for a hierarchy of entities: individuals, first-order predicates (which denote relations on individuals), second-order predicates (for relations on first-order predicates), and so on. Individuals and predicates may be variables, and expressions involving variables may be *bound* by the application of the quantifiers 'for all' and 'for some'. First-order logic with equality and functions (FOL) is widely used as a descriptive tool in mathematics; it permits no higher-order predicates, but is enriched with a special equality predicate and a set of function symbols. The use of second-order logics is not uncommon, but third-order systems are seldom mentioned.

---

[2]– quoted by Gries in (Dijkstra 1990 p229-236).

Modal logics enhance this framework with operators that express notions such as possibility and necessity.

There are doubts that FOL is appropriate for natural language structures. Israel & Brachman (1984) criticize reliance on FOL: "no significant fragment of any natural language has ever been semantically analyzed by way of a systematic translation into a standard first-order language". They favour Montague's (1974) approach, and see quantification as a particular source of difficulty. Hintikka's (1979) reflections doubt the usefulness of quantifiers in first-order semantics. His game theoretic approach to semantics of FOL and natural language (in Saarinen 1979) reveals that sentences with nested quantifiers are hard to interpret because they involve planning several moves ahead in a game. Despite the resulting difficulty in feasible computation with FOL, to a certain extent its theories can be converted into logic programs – PROLOG is based on Horn Clause Logic, a restriction of FOL that avoids explicit quantifiers.

An Algebra is a language of equations between terms that are constructed from function symbols and constants. Algebraic terms have tree-like structure and use equational logic which is easy to manipulate. As a logical language this is much less expressive than FOL, but has the advantage that it can be used as an executable specification language (e.g. OBJ3 – Goguen & Meseguer 1989).

Higher-Order Logic (Combinatory Logic and Lambda Calculus) can be regarded as a restricted form of equational logic, that has long been used as a meta-language for general computation. It is based on functions and functional abstraction, and gives rise to the Functional Programming paradigm. Israel and Brachman (1984) consider lambda abstraction[3] to be an essential technique in semantics.

Set Theory accommodates all these forms of expression. It has been the foundational language of mathematics this century, and is a clear candidate for formalizing general systems, as applied in specification languages such as Z and VDM. Its relationship with FOL is circular: it is normally

---

[3]Abstraction mechanisms are widely used to avoid (or at least hide) logical complexity. In computational terms, functional abstraction works by notating a game-winning strategy in place of mere assertion of winnability.

Lambda abstraction enables compound predicates to be formed. For example, **Pa & Qa** could be written **Ra**, where $R = (\lambda x.\ Px \ \&\ Qx)$. A similar construction exists in Set Notation:

$a \in P \wedge a \in Q$ can become $a \in R$, where $R = \{x \mid x \in P \wedge x \in Q\}$; i.e. $R = P \cap Q$.

presented as a theory in FOL, based on the 'membership' predicate ($\in$); the standard semantics for FOL is based (in turn) within set theory.

Category Theory (to be explored in Chapter 5) is an abstract foundational form of logic that provides a semantic approach to formal theories as *categories*, avoiding much of the tedious reasoning based on their internal syntax. One application is in studying algebraic theories; another is Topos Theory, which provides computational generalizations of set theory and a semantics for Lambda Calculus.

3.1.2.2 Denotational Semantics

A denotational approach to semantics interprets expressions into mathematical objects in a *domain*; techniques for this have been pioneered for programming languages (Milner & Strachey 1976, Gordon 1979). These domains may be complicated structures — e.g. directed-complete partial orders (*dcpo*) (Vickers 1989 ch10) *coherence spaces* (Girard *et al.* 1989 ch8) and *Chu Spaces* (Pratt 1995) — defined to resolve foundational issues in computing. This kind of interpretation is not necessarily a translation, because mathematical objects may be specified by their external behaviour, not how they are represented — as advocated in Wells (1994). It seems that in all these cases there is a postulated "space" that these denotations, or abstract classes of manipulations, belong to, and semantic analysis determines the place of each expression in this space.

A paper by Caswell (1997) compares three formalisms for programming language semantics. Denotational Semantics is shown to be equivalent to Action Semantics (Watt 1991) and Structural Operational Semantics (Plotkin 1981), for a particular target language. Caswell notes four uses of formal semantics: for language description, for checking compiler and interpreter correctness, for user reference manuals, and for reasoning about programs.

3.1.2.3 Pragmatics and Categorial Grammars

The idea that semantic aspects of language can be understood by logical analysis of its use in the intended context, i.e. *pragmatics*, was exploited by Montague (Thomason 1974). His work gave rise to *unification* grammars (Shieber 1986; Goguen 1988), which were developed in computational linguistics to interpret natural language into logic — the unification procedure is essentially a pattern-matching on algebraic terms, also used for example in execution of PROLOG

programs, and for function definitions in Functional Programming languages (Bird & Wadler 1988).

Development of these methods is currently an active area of research, with recent work on feature structures (Pollard & Sag 1994), and Categorial Grammar (Moortgat 1988, Wood 1993, Morrill 1994, Carpenter 1996), which derives from an elegant semantic calculus proposed by Lambek (1958). The intention (Carpenter 1995) is to create a general notion of grammar as constraints — a computational and logical system that integrates conditional constraints from all levels, phonological to pragmatic. Although these systems are idealizations of human language processing, he claims they have some affinity with psychological observations.

Glyn Morrill (1993) approaches language as an association between *prosodic* and *semantic* properties, linkings of form and meaning, where form is a bundle of properties; this derives from the semiotic tradition of Saussure, Carnap, Tarski, and Montague. The resulting linguistic programme attempts to specify language models for fragments of natural language. He describes the intuitions behind Categorial Grammar, which could provide a high-level logic of signs, a general framework allowing new fragments to be formalized and integrated.

Morrill reflects that empirical concerns have caused a trend towards 'lexicalism': the encoding of idiosyncratic information in the lexicon as the best way to formulate generalizations. Categorial Grammar has no syntactic component; it does not need to manipulate feature structures: just projection of lexical properties according to the interpretation of categorial operators. The syntax here is not in the data, but in the theory relating prosodics and semantics; in fact for categorial logic, it is the proof-theoretic meta-theory for the model theory or logic of the categorial operators.

3.1.2.4 Situations

An analysis of the pragmatics of language is provided by *Situation Theory* (Barwise & Perry 1981). Following this approach, Devlin (1991) has explored the possibility of basing a full theory of communicative acts on an abstract notions of situation and *infons* (items of information).

Barwise & Etchemendy (1988) consider reasoning to be the manipulation not of symbols, but of multimodal information. They argue for accommodating the complex features of real reasoning tasks (incomplete information, uncertain relevance, unknown conclusion), for which purpose they have developed an integrated approach to human reasoning with text and graphics combined, that directly addresses the situation in which an expression is 'uttered'. This is demonstrated in their

system (*Hyperproof*: Barwise & Etchemendy 1994), which is designed to help students learn to reason using both diagrams and the language of FOL. Hyperproof embodies a mathematical framework that considers models and propositions together; it formally describes both the syntactic and semantic domains in the same symbolism. It employs a situational calculus that has two binary predicates: a syntactic relation that a situation *supports* an infon, and a semantic relation that a situation *carries* an infon.

The topic of reasoning with text and diagrams is researched further in (Barwise 1993, Barwise & Hammer 1994).

3.1.3 Graphs and Graph Grammars

We next observe how graph rewriting has provided an important group of techniques, through a generalization of methods well known in linguistics.

3.1.3.1 Graphs, Attributes and Constraints

Graphs are a family of mathematical *combinatoric* structures, i.e. a graph is a configuration of objects such as 'nodes' and 'edges' (or relations). The family contains *digraphs*, *webs*, *hypergraphs* and many other types, though terminology for these is not fully standard. Graphs can be seen as a generalization of the notion of string that underlies formal textual languages. This leads to a common way of modelling a graphical notation as a *formal graph language*: a set of expressions that are instances of a specific type of *graph*.

For graph languages, Courcelle (1987a,b; 1990, 1994, 1996) has made extensive and detailed studies of specification by first- and (monadic) second-order logical constraint, and algebraic graph expressions (1987a, 1996), though his focus is not on graphical notation. From our point of view, his approach via Universal Algebra (Courcelle 1987a, 1996) has some relevance to editing of graphs, since it describes the process of generation by applying operations.

In operational treatments, graphs are often augmented with *attributes*, which amount to linkages between objects and value-spaces (e.g. number). Since these spaces are often infinite and highly structured, the mathematics involved is no longer confined to combinatorics. Constraints on attribute-values must be expressed in some logical language.

Nagl (1987) describes a software development environment which employs graphs as a meta-

notation for all structures, providing a uniform model for all problem areas. Graphs act as knowledge bases (one per document), and the graphs for a given class of documents belong to a certain type, whose structure reflects the requirements for processing those documents. He uses *attributed directed graphs* with labels on nodes (to describe the class of item), and edge labels to specify the specific relation which holds between two items. Attributes represent values which are determined at certain nodes or edges.

3.1.3.2 Grammars and Rewriting

Corresponding to textual grammars, *graph grammars* have been studied extensively since 1970, and several international workshops have been held (Claus *et al.* 1979; Ehrig *et al.* 1983, 1987, 1991, 1995). Schürr (1994a) defines a graph grammar as:

> "a system of productions that generates [from a start graph] a certain language of terminal graphs and produces nonterminal graphs as intermediate results. A graph rewriting system is a set of rules that transforms one instance of a given class of graphs into another instance of the same class."

As with textual grammars, generative graph grammars provide support for syntax-directed editing operations in the form of rewrite rules. They are able to express both context-free and context-sensitive grammatical constraints.

Graph rewriting generalizes both string rewriting and algebraic term-rewriting. Often, studies of rewriting are restricted to one particular graph type; in order to accommodate greater generality, many authors have resorted to the powerfully abstract language of Category Theory. In (Claus *et al.* 1979) and the tutorial (Ehrig, Korff & Lowe 1991), Ehrig formulates rewrite rules elegantly and generally, as double *pushouts* (DPO) in any suitable *category* of expressions (as will be described in Chapter 6). Most notations of interest fit well with Ehrig's notion, as do specifications of abstract data-types (Ehrich & Lohberger 1979).

Carradini & Montanari (1991) show how a hypergraph grammar of this kind can be converted into a term-rewriting system. Richard Banach (1996) has formally characterised graph grammars by forming categories whose morphisms are DPO rewrite-sequences. Bauderon (1995, 1996) is researching a general *pullback* method that encompasses DPO and other approaches, by describing parallel application of rewriting rules — a deterministic graph grammar can be described by a single rule that he calls a 'P-grammar'.

Implementation of rules entails a pattern-matching search, of a more general nature than the unification algorithms used in term-rewriting — and often much less efficient. Rules may be applied in parallel if they do not interfere, since their effects are local. The rewriting paradigm is also important as a candidate for modelling true concurrency in systems, and can be seen as a generalization of Petri-Nets (Corradini 1995).

The literature describes applications of graph grammars to software specification (Nagl *et al.* 1983; Engels *et al.* 1987), development (Nagl 1987), programming-language semantics (Pratt 1983) and diagram editing (Göttler 1983, 1987), for which practical prototype tools have been constructed for specific types of graph.

3.1.3.3 Attribute Grammars

Attribute Grammars (Deransart *et al.* 1988) are one approach to interpretation of formal languages — in particular those generated by context-free grammars, as first used by (Knuth 1968). These generalise to graph languages (Göttler 1983); the technique attaches values to nodes, and rewrite rules (attributed graph productions) specify how these values are to be updated, by means of formulas describing the evaluation rules for the attributes of its nodes. Thus the parsing operation is combined with computing the denotation of an expression. Attributes can also express layout features (Göttler 1987).

3.1.3.4 Computation by Rewriting Systems

Rewriting systems of any kind are known to provide general computing paradigms. In (van Leeuwen 1990 ch3), string rewrite rules are shown to give an effective system of (Church-Turing) computation. More generally, rewriting of terms and graphs has found application in design of practical computational languages.

A *Term Rewrite System* (TRS) is a set of rules which replace subtrees in tree-structured algebraic terms (Jouannaud 1985; Lescanne 1987; Dershowitz 1989; van Leeuwen 1990 ch6). Term rewriting has been studied in connexion with equational logic programming, for example OBJ.

Peyton Jones (1987) uses a graph rewriting technique, "graph reduction", for implementing pure functional programming languages. Graph Rewriting Systems (GRS) generally are the subject of

research by the European collaborative reseach project GETGRATS[4] (General Theory of Graph Transformation Systems), which aims to compare, combine and unify the various approaches to graph rewriting, and classify their expressive power. The related projects PROGRES (Schürr 1990, Schürr *et al.* 1995) and GRAS (Kiesel *et al.* 1995) successfully employ GRS as a core technique in a full software engineering support environment, as reviewed below (§3.3.3).

3.2 Formalisms for Graphic Notations

Having noted the basic theoretical approaches that we would expect to find applied to notational description, we now investigate some specific attempts at formalization.

Descriptive methods have mainly been devised for the purpose of providing a foundation for diagram processing tools. For instance, Göttler (1987) addresses the task as analogous to that of building a compiler for a new programming language, where standard definitions of syntax and operational semantics are necessary inputs to a compiler generating tool. For Rekers & Schürr (1995b), syntax definition serves to specify syntax-directed editing, can generate a graphical parser, and is a necessary precondition for semantics definition. Minas & Viehstaedt (1995) hold that diagram notations should be described by a formal model, to support an editor that can guide users in syntactic correctness of diagrams.

A variety of formalisms for defining graphical syntax of notations are reported in the literature; they may be divided into two main groups. The first group is based on systems of *logic*, and comprises specification languages, methods that focus on spatial properties, and algebraic semantics. The second group employs techniques of *graph rewriting* to define grammars and operations on notational structures.

Each subsection below ends by considering to what extent the reported formalizing methods are adequate, as regards our concerns here. We remark that most of these descriptive approaches have been developed in parallel with research for this thesis.

---

[4]GETGRATS is a research network funded by the European Community. The coordinator is Andrea Corradini at the Dipartimento di Informatica, Università di Pisa, Italy.

3.2.1 Specification Languages for Notations

We first review logical specification methods. Specification languages adapt more general logical systems to the specific problem of diagram definition. We find four textual languages (GDL, PSN, VCT and VODL) and a single example of a *pictorial* language for executable specification, which are described here.

3.2.1.1 Graph Definition Language

Expressions in GDL (Welland *et al.* 1990) are used by a tool to instantiate an appropriate diagram editor and checker. GDL is a textual language that includes hypergraph and enclosure notions, and first-order logic constraints. Despite this power, it in fact restricts the class of notations that can be defined. Its designers have avoided allowing a hierarchy of diagrams, which was felt to be outside notational syntax. They had to extend the language to cope with some of the notations they tried to specify.

3.2.1.2 Picture Specification Notation

PSN is a meta-language developed for formal specification of graphical notation (Hekmatpour & Woodman 1987), as a medium for driving graphics editors. It is a rich language admitting first-order logic formulae, set-theoretic notation, function definition and a query notation for binary relations, in a mathematical style.

It is thus broader than GDL in expressive power, and it enables specification of a refinement hierarchy of diagrams, regarded as an important feature. Its designers report that it succeeds in overcoming dificulties previously experienced with grammar formalisms. PSN is supported by a LISP-like symbol manipulation system called Kernel (Hekmatpour), coded in 'C'. The use of Hekmatpour's system Templa Graphica (1990) has been reported by Nlckerson (1995).

3.2.1.3 Visual Concepts: VC^T

A paper by Serrano & Welland (1997) describes the language VC^T, a textual formalism for specifying syntax and semantics of diagrammed modelling techniques such data-flow and entity-relation – the authors note that software companies choose to tailor such diagrams to their own applications. The formalism is aimed at the automatic generation of software design tools, and is specific in scope so that its specifications may be concise, clear and readable. Indeed, it is not expressive enough to capture spatial inclusion, abstraction and specialization. The language is

based on set theory, and uses predicate logic to express semantic constraints. A specification of Data-flow diagrams (DFD), with some simplifications, is given as an example.

The paper also reviews GDL and PSN; GDL is regarded as too expressive, and PSN does not clearly separate geometrical relationships from syntax.

3.2.1.4 Visual Object Definition Language

More recently, Üsküdarli & Dinesh (1995a, (&b)) discuss how to create a visual specification formalism and an environment for specifying the syntax and semantics of visual languages. They describe the language VODL, for use in generating visual editors — a constraint-based, declarative picture specification language influenced by (Helm & Marriott 1991) and (Wang 1995). The intent is to support only a visual Algebraic Specification approach to programming, however, and not diagrams in general. Even so, VODL has a large and complex signature, with many list-ordered and polyadic operations.

VODL describes the visual tokens (lexicals) and spatial relations that comprise 'lexical syntax'. VODL specifies visual elements (pictures) via units called *visual object definitions* (vods), built from:

> primitive vods (Point, Line, Circle, Text, Polygon; and Collection-of-vods),
> vod-operations (overlap, difference) that recognize emergent objects, and
> standard graphical operations over all vods (e.g. add a geometric constraint, set an attribute value).

This approach is extended to create a visual formalism for specifying the syntax and semantics of visual languages. The work attempts to extend algebraic specification formalism ASF+SDF (Bergstra, Heering & Klint 1989) which is successful for textual languages. The specified lexicals are incorporated in the visual *syntax definition formalism* (VSDF) via a mapping that associates syntactic constructs with lexicals (Üsküdarli 1995). The syntax specifies a context-free textual language, which is provided with an algebraic semantics in ASF. This has the advantage that algebraic specifications are easy to define and comprehend, and can be executed by orienting them as rewriting rules. Tools such as compilers, type-checkers, editors can be generated automatically.

A simple context-free notation for set algebra is used to illustrate this. Definitions are stated in a textual algebraic syntax, which is specified like a context-free grammar; non-terminals are *sorts*,

rules are *functions.* The authors observe that their choice of algebraic meta-level leads to an underlying tree representation, and that it would be interesting to consider a meta-language that handles general graph rewriting.

3.2.1.5 Picture Logic

Bernd Meyer (1992) aims to provide visual languages with an executable specification that is itself expressed in a visual language – with the long term goal of building visual compiler development environments. He believes that declarative specifications, while having a formally defined semantics, should display "an intuitive correspondence between description and object", and be flexible enough to support complex diagram languages.

To achieve this he develops a *Picture Logic* for reasoning about visual structures which is derived from Horn clauses, augmented to specify spatial arrangements. Spatial properties can be expressed with abstract example pictures. Picture Logic uses visual terms instead of facts to capture the spatial structure of described expressions.

A *picture term* is a directed acyclic bipartite graph. A picture language consists of a set of spatial object types and a set of relation types. Non-ground terms contain four flavours of variables: object, group, background and frame. Group variables are untyped, and can be bound to any connected cluster of objects. A term may contain a single background variable and a single frame variable. Unification of picture terms is similar to finding the maximal join of conceptual graphs in (Sowa 1984). During unification, only those objects are bound to the background that cannot be bound to some object or group variable, and only those spatial relations are bound to the frame in which objects participate that have been bound to different variables.

To provide an operational semantics he embeds the logic in standard logic programming by implementing a new unification algorithm, but this is non-deterministic and so introduces a new level of backtracking. He notes that the inherent non-determinism of picture matching causes problems in parsing. For more efficient parsing, less expressive Picture Grammars can be derived in the same manner as Definite Clause Grammars in Prolog (Clocksin & Mellish 1987). As for textual grammars, regular, context-free and -sensitive classes can be distinguished.

3.2.1.6 Other Approaches

General purpose specification languages can clearly be applied to the problem, and some

examples are found. Ince & Woodman (1986) have proposed to formalize the graphical aspects of development methods by using *semantic nets*. David Gee (1995) uses Z notation to specify syntax of Yourdon data-flow diagrams, Jackson structure diagrams.

An approach by Helm & Marriott (1991), Constraint Set Grammars, is mentioned above; their approach is based upon Constraint Logic Programming. Other Logic-based approaches can be found in (Marriott & Meyer 1996; Haarslev 1995), which are described later in this section.

3.2.1.7 Adequacy of Logical Specification Methods

These reports show considerable disagreement on which notions of syntax or semantics they choose to focus upon. The power, spatial concepts and specification notations can also be called into question.

The textual specification languages GDL and PSN are more powerful than is warranted for syntactic definition. Both attempt to accommodate a practical range of graphical forms, apparently with the result that their logic is powerful enough to express not only syntactic constraints but also semantic and stylistic constraints. VCT correctly aims at simplicity, but its scope is too restricted for notations in general – even though it employs powerful logical languages.

VODL/VSDF is too restricted by the choice of an algebraic formalism. Picture Logic is interesting as a synthesis of grammar and logic-based techniques, but requires something more powerful than Prolog programming to formulate a definition. The VODL/VSDF approach is interesting, though, in that it separates the formalisms for pictorial, syntactic and semantic structures; this is also intended in VCT.

The specification languages intended for defining graphical notations are themselves predominantly textual. GDL, PSN, VCT and VODL do not provide a diagrammatic representation of their rules. Picture Logic advertises a pictorial form, but this is largely copied from the notation being defined, augmented with some abstractions. The resulting logical circularity is only resolved by the underlying term-based representation. The pictorial form is merely a convenient visualization of terms, which can only be interpreted by someone familiar with the notation being defined. The visualization in VSDF is of the same kind.

None of these are based on any particular theory of diagrammatic structure, but are rather

experiments with certain styles of description.

3.2.2 Spatial Logic Approaches

Some authors argue the need for notation formalisms to be founded on the logical properties of space – approaches such as (Citrin *et al.* 1994) that are not based on graphical elements are seen as limited by (Haarslev 1996a). The next two formalisms do not simply refer to spatial relations, but incorporate properties of space in their logical bases. A third report attempts to be more precisely spatial.

3.2.2.1 Description Logics

Haarslev (1995) refers to the three main areas – formal models (grammars), semantics (e.g. declarative and logical), visual reasoning (spatial and temporal) – which are identified by Chang (1994). He proposes a new formal framework to unify these, based on *description logics* (DLs); these are subsets of FOL that exhibit *structured inheritance* (Brachman & Schmolze 1985). Haarslev's paper features spatial logic for describing qualitative relations between elements: points, lines or convex regions. Pictorial Janus (PJ), a visual language for concurrent programs (Kahn & Saraswat 1990), is successfully used to illustrate the method – which is to be fully detailed in Haarslev (1996b). [PJ is also treated by Gooday & Cohn (1996a), and (Müller & Lehrenfeldt 1994); see next and §3.2.4 below].

The framework, described in (Haarslev 1996a), uses a spatial logic for semantics of notations, based on research into reasoning with diagrammatic representations and spatial databases; it combines DLs with:

> (1) topology based on (Egenhofer 1991), with interior, closure and complement as primitive operators on point-sets (objects), that are used to define a complete basis of five binary relations on objects.
>
> (2) spatial relations based on (Randell & Cohn 1992), using the single binary relation *is-connected-with*, and a convex-hull operator.

A Description Logic is a declarative knowledge representation system based on inheritance networks; it amounts to a term-rewriting language that rewrites single unique term names. It is specified by a set of concept *terms*, a set of *roles* (binary relations between individuals of concepts), a set of *disjointness* assertions among concepts and roles, a set of concept

membership assertions for individuals, and a *terminology* to map names to specifications of concepts and roles. Concepts may be *primitive* (specified by necessary conditions) or *defined* (by sufficient conditions). He claims the notation is much more suitable for human or mechanical inspection than FOL; defined concepts are preferred to Prolog clauses that can only express sufficient conditions. It supports both parsing and constructing examples from specifications. DL systems automatically detect cycles in semantic specifications, which are known to cause problems (Haarslev 1995).

Haarslev takes elementary lexical tokens as primitives, forming the roots of a taxonomy (a specialization hierarchy) of defined concepts topped by semantic categories. Assertions and queries are used to state and retrieve spatial information about individuals. This taxonomy is claimed to help reasoning via *subsumption* relationships; it is more expressive than type-theoretic frameworks (e.g. Wang *et al.* 1995). It can deal with ambiguous grammars by computing every model satisfying the specifications — though algorithms can be NP-complete or even undecidable. He proposes that DLs be combined with concrete domains, to enable algebraic definition of concepts and take advantage of constraint logic programming.

He emphasises that different notations will require different definitions for objects and relationships. A generic editor *GenEd* for visual notations has been developed from this theory (Haarslev & Wessel 1996).

3.2.2.2 Region Connection Calculus

Gooday & Cohn (1996a) apply a Region Connection Calculus (RCC), originally developed for qualitative reasoning about physical systems, to the syntactic and semantic specification of visual languages for implementation and verification. They blame the difficulty found in traditional formalisms such as attribute grammars on the absence of a spatial vocabulary.

RCC is expressed in the order-sorted logic LLAMA, using a formulaic notation (Cohn 1987). The primitive *C* (x, y) holds when the closures of regions *x* and *y* share at least one point. From *C*, eight exhaustive and disjoint dyadic 'base relations' are defined. The paper takes as example PJ, which it claims is very naturally specified in RCC (details in Gooday & Cohn 1996b). PJ is made up of Strings, Lines, DirectedLines, ClosedCurves, their enclosing Regions, and PrimitivePictureElements that may be superimposed at the same region. The parsing of a picture

into primitive elements is not addressed.

The authors plan to formalize the execution-semantics of PJ by means of assertions and retractions of spatial relationships stored in a database. RCC expresses specification, parsing and execution in a common language, "breaking free from the need to use any non-spatial language"; this is contrasted with Haarslev's approach of translating to textual Janus.

3.2.2.3 A Complete Spatial Logic

Oliver Lemon (1996) highlights the importance of a deeper understanding of the formal semantics of spatial logics. He reviews several logics, and assesses them against an adequacy criterion:

> A logic is *spatial* only if it is equipped with classical or intended spatial interpretation(s) with respect to which it is complete.

He shows that many fail to satisfy, since they are incomplete – consistent sets of formulae have no models of the intended sort. These include RCC (Randell *et al.* 1992, Gotts *et al.* 1996, Bennett 1994, 1995), which aims to model qualitative spatial relations between regions of R3 in FOL.

Lemon notes that Modal Logics are known to be limited to capturing only positional constraints imposed by spatial structure; they cannot capture irreflexivity and intransitivity. By using the extended modal system of (de Rijke 1992), he provides a complete axiomatization of 2D space, presenting a modal logic of connected regions that obey the Kuratowski (1930) planarity conditions. The logic enlarges upon it has a modal operator ◊p meaning 'connects with a region where p holds'. The connection relation is taken to be symmetric, and all regions are distinct; ◊u means 'anywhere'. Isomorphic graphs are dealt with using iterated modalities.

3.2.2.4 Adequacy of Spatial Logics

The argument for using spatial logic contrasts with the linguistic tradition; we do not find acoustic or phonic logic employed in describing spoken language syntax. Is the difference due to the importance of graphical analogy?

Haarslev's work manages to bring together graphical, syntactic and semantic description, by a deeper analysis of diagram structure; in so doing it does not avoid complexity problems. Despite Lemon's criticism, there seems no clear benefit in imposing complete 2D spatial constraints on a syntactic structure that only makes partial use of them. The paper by Lemon is, however, important in considering the question of how properties such as planarity may be represented in

syntactic definitions.

3.2.3 Algebraic Semantics

Algebraic methods emphasize the analogical semantics of diagrams, Also included here are some methods of treating diagrams in a linguistic framework.

3.2.3.1 Order-Sorted Algebra

Wang & Zeevat (1996) criticise the picture specification language approach as not based on an understanding of cognitive use of pictures to give a better grasp of the application domain. They address cognitive issues of the analogy between picture and meaning by a notion of matching in a semantics based on order-sorted algebra (Goguen & Meseguer 1989). Both pictures and the application domain are described in an order-sorted signature. A match is enforced by a signature morphism, following the work of (Indurkhya 1992) on metaphor, and (Pineda 1990).

A picture description language consists of a *graphical signature* to provide symbols and a *graphical theory* to give geometrical meanings to them, in an algebraic institution (Goguen & Burstall 1984). The signature has a partially ordered set of sorts, a set of function symbols and a set of relation symbols. Functions are either *natural* (representing emergent graphical objects, e.g. overlap) or *artificial* (generating a new object in the picture), or attributes (e.g. length, colour). Wang & Zeevat assume that graphical inference is axiomatizable by a geometrical theory expressible over the signature; similar assumptions are made about the application domain.

An interpretation by metaphor is described as a partial mapping from graphical signature (G) to application signature (A): a *signature morphism* from a subsignature of G to A. This approach gives a semantics only for a single expression. A pictorial language will then be characterized as a set of picture algebras — items in an expression are constants in a picture signature. The paper does not make it very clear how this works. Deterministic FSMs are used as an example, but little detail is given.

3.2.3.2 Homomorphisms

Likewise, Gurr (1996) notes the common belief that specific representations share a similarity of structure with what they represent; he attempts to define such similarity precisely by means of homomorphisms and isomorphisms, generalized from their algebraic usage.

He models each of diagram and meaning as an α *-world*: a set of objects (*domain*) and a set of relations between these objects (*relation-set*) – a definition that "echoes or subsumes other descriptions" of situations and their representations. The mapping between two worlds – the representing and represented – is regarded as bi-directional. Again, a representation system will be a set of such maps, one for each representation-world pair.

Gurr distinguishes between *intrinsic* isomorphism (illustrated by an use of the spatial 'left-of' relation to represent integer ordering) and *extrinsic* properties commonly enforced upon a representational system [as syntax] – (see §4.4.3).

3.2.3.3 A Linguistic Approach

Treatment of diagramming as having truly *linguistic* content is attempted by only a few authors, who have an interest in diagrammatic reasoning. Pineda *et al.* (1988) describe an interactive interface (GRAFLOG), that treats drawings as a linguistic extension of text, using Montague semantics to interpret interactions (Pineda 1990). Ewan Klein (1987) describes a project to develop (in Prolog) a system to integrate natural language with graphics in knowledge base query and update, via common meaning structures. Klein aims to find whether drawings can be analysed like expressions of a language, with syntax and semantics. He identifies the basic syntactic constituents of drawings by reference to the communicative context (textual annotations and dialogue history), rather than by formal structure alone. This work uses Kamp's Discourse Representation Theory, a version of first-order logic with a novel treatment of quantifiers, pronouns and anaphora (Kamp 1981). Klein develops and applies a sorted logic (*inL*) for semantic representation.

3.2.3.4 Adequacy of Algebraic and Linguistic Methods

What is lacking in both the algebraic studies is an appreciation of analogy as a *systematic* process within a notation as a whole. Without this, there is little we can do to untangle the the problem of how expressions are understood and used to aid thinking. The algebraic approach of Wang & Zeevat brings the graphical structure into relationship with its semantics, but ignores the need for a systematic syntax.

Klein's and Pineda's work moves into a broader area of analysing diagrams in a linguistic context; although this is attractive, it takes us beyond the scope of this thesis.

3.2.4 Grammars for Notations

We would hope that the generalized grammar-based approaches above (§3.1.3) would smoothly accommodate description of graphical language. Unfortunately, the reports show that this is not so. Graphs introduce problems that are not found in string or term rewriting. Once again the representation of spatial properties appears as a concern. A grammar for translation is reviewed, followed by some research that aims to locate grammars within a general hierarchy according to expressive power.

3.2.4.1 Difficulties in Grammars

Woodman *et al.* (1986) have examined the feasibility of using three types of grammar for specifying software engineering notations. Tree and *web* grammars were dismissed in view of complexity and parsing problems; *plex* grammars were found more successful, but still not sufficiently general. They proposed a grammar based on EBNF production rules, augmented by a set of relational axioms to specify which kinds of node may participate in which relations. They conclude in (Hekmatpour & Woodman 1987) that formal grammars have inherent difficulties with expressiveness, parsing, and dealing with incomplete diagrams.

Göttler (1987) uses *programmed* grammars that allow one user-action to be modelled by a program of productions, which are themselves drawn as diagrams. He reports the problem of their inability to express mutual constraints on attributes, as found in certain aesthetic requirements — though satisfying them is algorithmically NP-hard.

Courcelle observes (1990) that graph grammars are worse-behaved than string grammars, and they support no good notion of graph automaton, i.e. finite machine that decides whether a graph belongs to a given graph language. Other difficulties and limitations of various grammar approaches are described in (Wittenburg 1993). According to Schürr (1994a), the common belief that graph rewriting systems lead to inherently inefficient implementations, since many graph algorithms are NP-complete, is no longer well-founded. He considers that the situation is gradually improving.

3.2.4.2 Embedding

A central difficulty in defining graph rewriting is known as the *embedding problem*, described by Rekers & Schürr (1995b):-

> When a nonterminal is replaced, how does the production establish relationships between its context elements and the elements of its replacement?

They analyse three approaches:

1) Implicit embedding. Picture Layout Grammars (Golin 1991b) and Constraint Multiset Grammars (Marriott 1994) do not distinguish between vertex and relation objects. They thus have to constrain attributes in order to express relationships between objects; the embedding rules are a side effect of attribute assignments. The users may not be aware of these effects, and parsing is complex.

2) Extended context. The most readable solution is to embed new objects by extending left and right sides of a production with explicit context. In this case it is difficult to rewrite symbols that participate in a variable number of relationships.

3) Embedding Rules. A more powerful and convenient method uses separate rules that redirect a set of relations to their new context – as in some graph grammar techniques (precedence graph grammar: Kaul 1982), (Rozenberg & Welzl 1986, Ferrucci *et al.* 1994). Not only are these hard to understand, but all known parsing algorithms are very inefficient, unless production sides are severely restricted.

With regard to these issues, they tabulate the properties of eight approaches, and find all of them to be inadequate for defining the language of process flow diagrams. Their own approach (see §3.3.3 below) succeeds because it can handle context-sensitive productions that replace more than one non-terminal.

3.2.4.3 Relation-Based Grammars

A general notion of grammar need not be based explicitly on graphs. Grammars based on relations have been studied and applied by several authors (Wittenburg & Weitzmann 1996, Ferucci *et al.* 1996).

Wittenburg & Weitzmann's Relational Grammar formalism was first proposed as an extension to unification grammars of computational linguistics, for the purpose of efficient parsing. Grammars generate expressions of the following kind:

> An *indexed multidimensional multiset* consists of an indexed multiset of symbols (for graphical items), and a sequence of relations on these symbols (relations refer to items and their attributes).

The paper defends the choice of context-free syntax, despite the fact that it may not be powerful

enough to represent many visual languages. The work shows that such restricted grammar frameworks "can play useful roles in interfaces without having to represent the visual language in its entirety."

> "If we cannot demonstrate that a weaker formalism is useful in applications within its power, then how can we possibly be convinced that more powerful formalisms are of practical value given their known computational complexity?" Wittenburg & Weitzmann (1996)

We note that a relational grammar is presented in a formal textual syntax, with a clear but informally presented graphical equivalent, which borrows the graphical items of the language being defined.

A paper by (Ferrucci, Tortora *et al.* 1996) discusses features of visual language generation and recognition, and the goal of an uniform framework, from the viewpoint of another kind of Relation Grammar. The model provides a high-level description of an expression as a set of symbol occurrences (*s-items*) and a set of relational items (r-*items*) over s-items. An uniform mechanism is defined to rewrite both r-items and s-items by means of context-free production rules. The model aims to specify relationships among symbols at a level of abstraction that is less dependent on the underlying implementation of a graphical interface. The authors believe that an accurate analysis of the expressive power of visual grammars is necessary to fully exploit the capabilities of such a formal model.

The paper points out both analogies and differences with respect to other existing models; the main difference with the Wittenburg & Weitzmann approach is that r-productions can define constraints over composite objects in terms of relations between their components. In a further paper (Ferrucci, Pacini *et al.* 1996), relation grammars are compared with generative graph rewriting formalisms, and some equivalences between classes of grammars are established.

3.2.4.4 Spatial-Relation Grammars

Müller & Lehrenfeldt (1994) provide a case study on the language Pictorial Janus, using a version of Ferrucci's context-free relational grammar, which they claim is more readable and simpler than other constraint-attribute grammars. Each rule rewrites a single symbol as a multiset of symbols, subject to a set of topological constraints between sequences of terminal symbols (which are graphical objects). The authors also wish to investigate free-hand recognition (Zhao 1993).

We can analyse the formal structure created by such a grammar as a hypergraph whose edges join two sequences of nodes. In their example, they use three binary constraints (external-touching, inside, internal-touching), and one multi-ary (for spatial separateness of a set of nodes). This last requires hyper-edges – rules would be cumbersome if the formalism allowed only simple two-ended edges (only pairwise separateness).

Adjacency Grammars (AG) are used by Jorge & Glinert (1995) who extend them as a foundation for interactive parsing and handling partial input. Their work is related to Wittenburg's unification-based approach (Wittenburg *et al.* 1991, Weitzmann & Wittenburg 1994). Complexity is considered for distance-bounded adjacency languages, showing how *spatial enumeration* data-structures support efficient parsing.

The authors observe that associating constraints with productions provides a general control mechanism for parsing, enhancing the power of declarative semantics. They review two approaches to parsing of visual sentences that focus on spatial relations as a main component in syntax analysis. Golin (1991a) shows that parsing arbitrary attribute multiset grammars is NP-complete. Using dynamic programming he has developed an $O(N^9)$ off-line parsing algorithm for a subset called Picture Layout grammars (PLG). Marriott's CMG approach (1994) addresses the role of spatial queries, but precise complexity bounds are not given.

AGs extend PLGs with Adjacency and unbound productions (that establish logical aggregation). An adjacency relation is a constraint with an associated query function returning the *neighbour* set of a given visual symbol – a way of strongly grouping elements in a production. They achieve efficient parsing, refining Golin's (1991a) spatial operators to enable fast retrieval of candidates.

Adjacency is a "powerful and intuitive concept". There are three main types:

Algebraic:	A adjacent to B "if there is nothing in between";
Spatial:	indicated by geometric distance; (closeness);
Logical:	non-spatial, indicated by same-labels, or recursively formed lists.

Each production rewrites a non-terminal N, conditional upon an adjacency constraint over the attributes of the produced symbols, and synthesizes attributes of N. The parse result is a rooted directed acyclic graph (RDAG).

Directed graphs are used as example. By using contextual symbols they are able to parse graph structures without separate explicit constraints on nodes as arc endpoints – this leads to "non-tree

branches", as in PLGs and CMGs. At the top of their grammar, productions occur that are *unbound* by geometric constraints – e.g. those that generate a set of separate nodes.

The advantages claimed are a simple algorithm with easily specified visual grammars and easy parsing. They describe in detail an efficient on-line parse algorithm based on dynamic programming, alternating top-down / bottom-up operation; its near-linear complexity improves over purely syntactic methods that are quadratic.

3.2.4.5 Graph Grammars and Translation

Schürr (1994b) addresses the problem of using graph rewriting in translating graphs of one type into graphs of a different type, e.g. program syntax trees into control-flow diagrams. He argues against embedding source, target languages and the intermediate correspondences in a common superstructure, because this entails extending directed graphs with second-order relations; and needlessly preserving fine grain correspondences. He finds it more appropriate to use a pair of morphisms from correspondence graphs to source and target graphs.

A Triple Graph Grammar (TGG) is a purely declarative specification of translation, that can accommodate Context-Sensitive productions and many-to-many relationships. Their *correspondence graphs* and rules record information about the transformation process needed to propagate incremental change. These derive from Pair Graph Grammars (Pratt 1971) that translate strings to graphs, but are restricted to context-free productions and 1-to-1 correspondences between elements of data structures.

The notion is presented simply for unlabelled directed graphs, with notes on how to extend it to more practical cases. Each triple production depicts how the correspondence between source and target is maintained during a rewrite. In translating, it is necessary to parse a given graph L to yield a left derivation sequence and then apply the related sequence of right-productions to a start graph to give result R in the target language. To simplify, Schürr considers only monotonic productions, specified with a single pushout, so that a graph contains its own derivation history. For this case he develops a terminating translation algorithm; the result of a rewrite is proved to be unique (soundness), but completeness requires backtracking to find all left-derivations.

3.2.4.6 New Hierarchies

We have seen above that Ferrucci is concerned with comparing the expressive power of different

formalisms. Jorge & Glinert (1995) believe there can be hierarchies of formal visual languages according to the expressive power of spatial and logic constraints. Haarslev (1996a) also intends to build a complexity hierarchy of visual languages in future work.

Marriott & Meyer (1996) attempt to define a 'Chomsky hierarchy' of languages based on Constraint Multiset Grammars (Marriott 1994). These are chosen not only because of their generality, but for their close links with constraint logic programs. The authors have shown that certain other formalisms can be mapped to CMGs: Positional Grammars, Relational Grammars and Unification Grammars.

CMGs rewrite multisets of typed attribute symbols, and form a computationally adequate system if simple arithmetic on attributes is permitted. In order to reduce their power, the authors restrict attribute manipulation to copying, resulting in "copy-restricted" or CCMGs. Marriott & Meyer examine the hierarchy of expressiveness formed by analogy with type 0,1,2,3 string grammars. The power of CCMGs can also depend, however, on the complexity of first-order logic (FOL) formulae used to state constraints; this leads to a taxonomy of nine types. Parsing of these types is investigated and found to be expensive in complexity.

3.2.4.7 Adequacy of Graph Grammar Methods

We note that most of these descriptive techniques fall in between the difficulty of either being too powerful and impossible to parse, or feasible in parsing but too weak to cover all the notations desired. Marriott & Meyer (1996) find that the graph grammar approach is generally deficient in arithmetical or deductive treatment of spatial relations and their interdependencies. This is overcome by means of logical constraints to control productions.

Efficiency is then an inevitable concern, given the difficulty of parsing powerful grammars. Although researchers have attempted to improve matters, many do not make the connection that this can ultimately only be achieved by reducing the power of the grammar to a minimum. The paper (Wittenburg & Weitzmann 1996) confronts this most clearly by suggesting that a simple grammar can usefully deal with *part* of a visual language syntax. The above attempts at organization into a hierarchy of strengths are thus a very welcome (recent) development. A hierarchy provides a way to reduce or at least control the parsing problem; by dividing into partitions, each with different power in the hierarchy, the part of the syntax that requires a more

complex grammar may be minimized. No authors, however, include in a hierarchy the logical specification methods reviewed earlier in this section.

We ascertain that Adjacency Grammars (Jorge & Glinert 1995) make a helpful contribution to efficiency, since the technique of "spatial enumeration" exploits advantages of 2D layout, corresponding to cognitive factors (ease of visual search) that are believed to be important in diagrams. There is no reason to believe that parsing should in general be feasible if such graphical constraints are ignored.

The reports do not pay much attention to how the grammatical specifications are notated, being more concerned with their properties. Although the graph grammars are not specific to pictorial structure, their rules often invite a diagrammatic form of expression, based on the standard ways of drawing graphs. The pictorial forms of graphical elements can be incorporated into expression of rewrite rules, for the purpose of illustration. It is less clear how relational grammars or constraints within grammars might be diagrammed.

3.3 Software Tool Support for Notations

The various kinds of notation processing needs in software development methods are served by a range of computer aids or tools. This section remarks upon the notational functions required of tools, and gathers reports of several that employ formal techniques such as those outlined above. In the reports, several authors have assessed the tools available or have described their own developments. We now wish to discover how successful and versatile these aids are in their capability for processing notation in general. In this instance our principal concern lies with the suitability of any theory that underlies the tools, rather than their overall performance.

3.3.1 Requirements for Tools

We start by examining the capabilities that tools must possess in order to support notation in software engineering. The previous chapter (§2.2.1) mentioned many roles for notations, e.g. knowledge representation, formal specification, structural diagrams, logical calculi, programming codes, users' specialisms and project management. Here the kinds of processing required to assist the main roles are noted, and in particular the properties of notation editors are looked at in some detail.

3.3.1.1 Notation Processing Requirements

We find two views of the place of notation in the software development context. In the first, expressions are seen as part of a growing documentation that records the current state of the project, and at least consists of a partially determined system description. Documents contain a set of views; when complete, all relevant knowledge of the system's behaviour may be deduced. Tools for browsing and understanding a notated document may need to translate into styles and views that suit the viewer.

The second is an interactive view: for instance in (Ince & Woodman 1986) project knowledge is recorded in a database that expresses replies to queries using required notations. These replies are not simply extracts, views or translated forms from a static document, but are actively constructed by logical inference. The approach of Goguen & Meseguer (1987) is to regard a specification as an inefficient program; when a logical query is made about the system specified, it is answered by means of a deductive process. In the same way, executing a program effectively answers the precise "query" as to what output the system will give on receiving a given input. More generally, in (Cohen *et al.* 1989), project documents and data would afford access by a pattern-matching, browsing mechanism, with general heuristic reasoning mechanisms to support inference. Knowledge is expected to be encoded in different logics, whose rules must therefore be available as parameters to the access process.

In any case, notation tools support an *User Interface* to the documentation, enabling a participant to assimilate or extend the information in the system description. Queries could be answered in a notational style chosen by the user. There are two sides to this communication: output expressions must be automatically produced and presented; input expressions must be interactively composed and edited. Ince & Woodman (1986) have developed software ("Toolbuild") that is designed to support textual notations in this manner. They determine that tools must have facilities for storing and retrieving information in a project knowledge base, indicating a need to display structures pictorially, with editing assistance. Editing should involve checking syntactic correctness in terms of a formal definition, and performing some semantic checking. These facilities must be versatile enough to interface with existing and future tools.

Such tools require a high level of abstraction. A requirement for some *metalogical* means for relating different notations is noted by Black *et al.* (1987) who have developed a unified semantic

model using a frame-based representation, in order to integrate the many contemporary development methods.

Cohen *et al.* (1989) consider automated aids for operating on specifications:- they require composition and editing of expressions, directed by syntax and semantics; and the ability to perform symbolic manipulations on system descriptions, such as theorem proving, simulation and execution. To fulfil these they envisage generic techniques for processing notations, or *metatools*: various symbolic manipulators, parametrized by the syntax and semantics of the languages.

3.3.1.2 Editors

Amongst these needs, we focus upon editors, that are clearly essential. What should we expect of a tool that facilitates the editing of expressions?

We have some expectations owing to the familiar processes of editing and interpreting in respect of textual programming languages. In any graphical notation, creating expressions is a matter of making marks on a screen, mediated by software that both enables changes and restricts freedom. Text, for example, allows only shapes from a fixed character set to be displayed. The system's interpretation of the marks can play a part in guiding the user's input towards an acceptable expression.

The early text editors used for programming provided no help with syntax; errors were detected and indicated by a separate parser during a subsequent compilation stage. Syntax-directed approaches avoid parsing by using a generative grammar to support insert/delete operations directly — but grammars do not enable copying of subexpressions nor deletions from lists, for example (Gruzlewski & Weiss 1991). These operations require a 'derivation structure' to be stored and subjected to transformational rules.

A paper by Kent Wittenburg & Louis Weitzman gives a useful discussion of the problems of generic editors for visual languages:

> "It is naive to think that satisfactory visual language Interfaces can be implemented using generic graphical editors combined with an analog of YACC to interpret graphics. Unlike generic text editors, visual language editors must be specialized to the graphical language at hand."
> (Wittenburg & Weitzmann 1996)

Thus *Visual Language Interface* toolkits cannot afford to ignore the relationship between the

language of user gestures and underlying representations of graphical objects and relationships. The authors consider approaches to the question of what ordering an user might follow when constructing an expression; one answer is to define mappings from the underlying grammatically-based descriptions to procedurally-defined editors for creating these descriptions (Backlund *et al.* 1990).

Wittenburg & Weitzmann's reported experience with some users of their editor indicated that the enforcement of a strictly hierarchical visual syntax was a matter of controversy. Some users "wanted to be able to informally sketch process diagrams, particularly at early stages of a project ..." – requiring a language of general directed graphs that is not context-free.

3.3.1.3 Flexibility and Guidance in Editing

The question of how much guidance to give in editing is mentioned by several authors. Some (Coomber & Childs 1990, Gruzlewski & Weiss 1991) stress checking and maintaining semantic correctness during editing. Gruzlewski & Weiss address the problem of structural editing of program texts. Normal semantic checking is criticised as insufficient in that corrections may introduce new errors. They develop a syntax-driven editor using reversible grammar rules, that works directly on the derivation tree of an expression, in order to hold semantic correctness as an invariant. The prohibition of incompatible or inconsistent expressions, however, is likely to conflict with the need to store partial diagrams, that is noted by Hekmatpour & Woodman (1987).

A *syntax-directed* editor prohibits the drawing of syntactically incorrect diagrams (Göttler 1987). In contrast, Welland *et al.* (1990) feel that users should be offered a choice of how permissive or directed this syntactic control of editing is to be; and McWhirter (1995) says that syntax-directed editors may overly constrain how a user interacts with the application. Rekers (1994) notes the inadequacy of a pure syntax-directed approach, and asserts the necessity of offering users freedom in how they develop diagrams, with structured support on demand.

3.3.1.4 Specific and Generic Tools

Some tools are intended only for supporting a specified few notations. Specific notations are supported within the context of a particular methodology: CASE tools that have facilities for processing graphical notations exist for methods such as Yourdon Structured Design (Yourdon & Constantine 1978), JSD (Jackson 1983), SSADM (Gane & Sarson 1977), HOOD, HOS (Martin &

McClure 1985 ch38), MASCOT (MASCOT 1987).

Generic tools can support a certain range of notations, when provided with suitable specifications for them. Such tools vary in restriction on freedom that users are offered in their choice of syntax. In a project to provide syntactic support for graphical notation tools, Hekmatpour *et al.* (1988) report experimental evidence that users sometimes require maximum flexibility in syntactic style and layout, preferring to modify syntax to suit their own conventions. This would require users to understand and modify notation specifications, or to indicate their preferences somehow by interacting with the tool.

Both editorial guidance and the division between generic and specific operation are observed by Welland *et al.* (1990), who characterize tools according to three or four oppositions:

> stand-alone / integrated (within the development method)
>
> method-specific / configurable (for user's notations)
>
> syntax-driven / permissive;
>
> Checking may be:- off-line (global)/ interactive (incremental).

A *configurable* tool can be modified for a choice of diagram syntaxes; it is generic.

They describe two of the better tools then available for generating and editing diagrams. Firstly, the method-specific tool *Analyst* is assessed. It uses a permissive-interactive style of editing, with checking coded as Prolog rules. Since diagram syntax is coded in PASCAL, it cannot be easily modified. Secondly, they assess *MacCadd*: a configurable stand-alone tool with an interactive editor, that expresses syntax rules in Prolog, though the choice of syntax is very restricted – there is a fixed vocabulary of symbols. They comment that Prolog is unsuitable for integrating into design method software.

3.3.1.5 Assessing Editors

The extent of liberty and guidance offerable by an editor depends on the depth of interpretation undertaken by software during the editing process. This in turn depends on the extent of the formalization of notation structure on which it is based. We may ask:

> Does the formalism cover geometric and semantic details as well as abstract syntax?

If a tool is well constructed, there is a separation of various concerns, to allow flexibility. We may, for instance, ask:

Can the changes to editing style be made independently of the choice of notation?

Can the notation syntax be changed independently of the development method?

The power of the specification formalism determines how complex the syntax is allowed to become.

How easy is it to make changes to syntax?

3.3.2 Reviews of Notation Editors

We seek to discover whether graphical tools are general and flexible enough to be easily integrated into any given software development method. Though we are mostly interested in designs for *generic* editors and other tools, we begin with mention of a few tools for specific notations. The review here looks at the representations used for expressions, syntax specification and editor specification.

The design of such tools is a cause for concern. Minas & Viehstaedt (1995) find that some tools support simple kinds of diagram, but only very few systems for generating a diagram editor are based on a formal model.

3.3.2.1 Some Specific Graphical Tools

Coomber & Childs (1990) have an object oriented editor & simulator (in Smalltalk) for graphical prototyping of Real Time Systems, specific to Transformation Schemas (Ward 1986). It verifies syntax, assists in proving semantic correctness. Editing consists of placing nodes and connectors; before a connexion is made it is checked for allowability, using menus to present available choices.

Jensen (1991) discusses properties of editors for coloured Petri Nets as a language for system design. Standard ML was used for formal underpinning.

I-Pigs is an interactive graphical environment for concurrent programming, developed by (Pong 1991). It guarantees that the graphic program is syntactically and semantically correct, and can display execution.

Ludwig2 is a general purpose event-driven visual programming language (Pfeiffer 1995). It uses three modes of expression: graph manipulation, arithmetic and user interaction, but is based on a common processing model of algebraic graph grammars; program and data are represented as

hypergraphs (i.e. arcs may link subgraphs as well as nodes).

Citrin *et al.* (1995) describe a Visual Lambda Calculus VEX that is designed to be easier to understand than the textual calculus; it is a component of the object-oriented VIPR language.

3.3.2.2 Some Generic Editors

McWhirter (1995) reviews some extensible graph editors, which "provide support for a narrow domain of languages and interfaces ... typically have a set of predefined language constructs (e.g. node, edge and graph)":-

> *Garden* (Reiss 1987) contains a tool *GELO* that allows definition of graphical representations of sets of typed objects, with a greater range than the others mentioned; it allows for text and tiling as well as graph layout.
>
> *EDGE* (Newberry & Tichy 1988) is a generic graph editor with focus on automatic layout, graph abstraction and extensibility. 'Subgraph abstraction' groups a set of nodes within a parent node (visual containment). 'Edge-concentration' groups sets of edges with common source and target, via a special node. Extensibility is provided by an object-oriented approach, but this is limited.
>
> *PRONET* (Sylva *et al.* 1991) is a generic graph editor for development of graphical network modelling, based on GNMS, a general descriptive mechanism for structural aspects.
>
> *LOGGIE* (Bolognesi *et al.*1991) uses attribute grammars. Editing commands take the form of complex derivation functions applied to the abstract syntax tree that represents an expression. Each node of the tree can have any number of graphical representations, called *aspects*; constraints can determine how a child node is placed with respect to its parent node, and nodes in the tree may be linked by *garlands*.
>
> *Pallette* (Golin *et al.* 1992), based on a picture layout grammar, uses a *picture parsing* approach that is criticized by McWhirter as providing support only for graphic constructs, not language constructs, thus creating a semantic gap.

Two papers (Welland *et al.* 1990, Beer & Welland 1987) introduce a general purpose tool, *ECLIPSE*, which can be integrated with a project support environment to handle all its graphical notations. Its design aims for a configurable, permissive-interactive tool that accommodates to syntactic and semantic checking specified by the user. ECLIPSE uses the specification formalism GDL referred to in the previous section (§3.2.1). McWhirter (1995) assesses this tool as less restricted than most – it provides a graphical editor to define the basic representations of the graph objects.

One of the earliest attempts is by Göttler (1987, 1990); he describes the process of eliciting

diagram knowledge from a user, and designing a Programmed Attribute Graph Grammar (PAGG) interface for syntax-directed editing, implemented in LISP. Design was found to be fast compared to ad hoc programming of an editor. Minas & Viehstaedt (1995) find, however, that the PAGG layout of diagrams is troublesome and editing is inconvenient.

Clement *et al.* (1990) describe *CENTAUR*, a generic interactive programming environment parametrized by syntax and semantics of programming languages. Tools generate a structure editor and an interpreter / debugger within a uniform graphical interface. Their software separates graphics from behaviour: Geometric graphical objects are treated as reactive systems that change state in response to input events and generate output events; their behaviour is written in the Real Time language *ESTEREL*.

Ballance *et al.* (1990) present *Pan*: a language-based editor for integrating development environments, based on a context-free Logical Constraint Grammar (LCG) – a grammar that annotates its symbols and productions with goals expressed in Prolog. These specify constraints on the language generated by the grammar. LCGs were successful in solving problems of scoped variables, but the authors found that modifications were needed to the Prolog model to make them practical.

Garnet (Myers *et al.* 1990) explores the use of constraints for graphical user interfaces. The structure of valid diagrams is more or less hidden, however, and must be maintained by the programmer – according to Minas & Viehstaedt (1995).

Rekers (1994) proposes to implement graphical editors that allow both structured and free editing by parsing diagrams in two phases, corresponding to graphical structure described by a *spatial relations graph* and syntactic structure described by an *abstract syntax graph*. He considers this distinction to be very useful. Parsing makes use of the graph grammar formalism PROGRES (Schürr 1990, 1994a). Although the results were positive for the very simple graphical language considered, he concludes that it is unclear whether more complicated graph grammars can be treated as easily.

Haarslev & Wessel (1996) are developing *GenEd* – a generic semantic editor for formal reasoning about visual notations (see §3.2.1 above).

Further reviews of research on syntax-directed editing of text and diagrams, and visual language

parsers, can be found in the thesis of Viehstaedt (1995).

3.3.2.2 Guided Editing of *Visual Objects*

Serrano (1995) purports to show why current tools supporting diagramming notations are not satisfactory. He aims to provide non-obtrusive guidance and allow flexibility in editing through the management of partially constructed expressions, an approach that he outlines but does not formalize. Entity-relation diagrams are used as an example, with constraints expressed in natural language. In acknowledging the need for a formal approach, he proposes constraints in FOL, executable by Prolog.

He notes two extreme approaches to the question of guidance: (1) the editor offers none – when there is a separate phase for semantic evaluation (normally the diagram is translated into a textual version and then parsed), or (2) the editor "shepherds" the user through the editing process – evaluation is carried out during drawing and inconsistency is forbidden.

The solution that Serrano proposes is to embed all the semantic constraints in the editor, so as to allow automatic diagram validation without limiting the user's freedom. A diagram is composed of *Visual Objects* (VO) that have a logical part and a physical part, and are either *icons* or *connections*. 'Semantics', which he defines as VO behaviour during the editing task, is expressed by constraints. A VO has three possible states: *Complete*, *Accepted* or *Disconnected* according to its degree of incorporation into the parsed structure. Diagrams may be *Valid*, *Inconsistent* or *Wrong* – Inconsistent diagrams are merely incomplete, but Wrong ones require backtracking to correct. The constraints enforced on a VO depend upon its state as well as its spatial relations.

The four steps for editor design are (1) Identify VOs, (2) express their behaviour in terms of constraints, (3) Identify those constraint violations that would cause a diagram to enter a wrong state, and (4) Define any compound commands needed – as design options that enhance usability.

3.3.2.3 *VisualGen*

Chok & Marriott (1995) describes a parser generator (VisualGen) and a graphics editor which generate a sophisticated user interface from a CMG. It features quick incremental parsing with geometric error correction via a metric space. The editor allows manipulation of diagram components – maintaining constraints to preserve 'semantics' [i.e. syntax].

Unlike attributed multiset grammars or relational grammars, CMGs allow negative constraints and thus enable deterministic parsing. The graphics editor is not syntax based, but is appropriate for a hand drawn approach. Constraints are generated automatically during parsing – diagrams can be drawn in any order.

3.3.2.4 *DiaGen*

Minas & Viehstaedt (1995) have recently addressed much the same problem area as this thesis. They describe a generator (*DiaGen*) for diagram editors that is supported by a formal method: hypergraph grammars, which they claim to be simpler than graph- or constraint- grammars. DiaGen is a fully functional system that has been tested with large specifications.

A hypergraph grammar describes the structure of diagrams (e.g. as Harel 1987 uses for Statecharts) in a "much more intuitive and advantageous model for diagram representation" that permits direct representation of multidimensional relationships, as needed for layout. Context-free hypergraph grammars rewrite edges, initiated by a *starting graph*. The total system is a "highly flexible method" of diagram representation.

The user sees only valid diagrams, mapped to the screen from terminal hyperedges of an internal *derivation* hypergraph. A terminal symbol image is composed of primitive elements (lines, text, etc.) – a hyperedge connects (*visits*) a fixed number of nodes, which stand for points, fixing its position. Both edges and nodes carry attributes, and node attributes apply to all visiting edges; as a result, hypergraphs require few constraints. Editing is carried out by direct manipulation of diagram parts, to avoid concerning the user with grammar rules; an incremental algorithm adjusts layout. Layout conditions are attached to productions as multidirectional constraints, which may be linear inequalities.

The examples given, NSD & Flowcharts, are found to have similar specifications, differing only in terminal-edge mapping and constraints. (Productions are shown graphically, with corresponding nodes labelled by letters.) The authors are contemplating the use of context-sensitive grammars, needed for other examples.

Edit modifications are specified by *transformations* – compound transitions from one set of diagrams to another, that operate on the derivation tree. This method also supports animation (execution) of diagrams. In practice, transformation specifications are the major part of the work.

In order to avoid this need and extend DiaGen to allow arbitrary manipulations, they are constructing a parser that can efficiently identify maximal syntax-trees, find inconsistent or invalid parts, and suggest completions. A designer should then be able to create an editor in a few hours – ideally having 'drawing-tool' behaviour interpreted by a parser.

3.3.3 Visual Programming Tools

Lastly we look at attempts to support visual programming, an area which suffers from a lack of tools (Üsküdarli & Dinesh 1995a). Together with the complexity of language representation and structure, this lack has prompted considerable work on generating visual programming environments (VPE). Rekers & Schürr (1995b), comment on the lack of tools that have efficiently working parsing algorithms.

3.3.3.1 An Algebraic VPE Generator

Üsküdarli & Dinesh (1995a) propose a VPE generator environment which they have not yet implemented. Their paper discusses generation of visual editors and VPE generation based upon visual language specification in the picture definition language VODL and syntax formalism VSDF, reviewed above (§3.2.1). The editor-generator yields a tool for the construction and execution of visual programs (Üsküdarli 1994), within an algebraic framework; programs are executed by term-rewriting.

3.3.3.2 *PROGRES*

Schürr *et al.* (1995a) report on the multi-paradigm language PROGRES, mainly used for specifying abstract data types. It has the flavour of a visual database programming language with powerful pattern matching, replacing facilities, and recursion — claiming to be the first rule-oriented visual language which has a well-defined type-concept. The system provides an integrated set of language-specific tools to support intertwined editing, analyzing, browsing and debugging of specifications as well as generating prototypes. It is descended from a whole family of (programmed) graph rewriting languages. The underlying nonstandard database system is GRAS (described below).

PROGRES has context-free syntax, and dynamic semantics — though not especially tailored to parsing diagrams. Its advantages over other VPEs are its strong typing, and provision of data

definition sub-languages (not just manipulation of data structures).

The paper discusses the syntax-directed editor and its incremental type-checker, with the running example of control flow diagrams, and recognizing the absence of 'go-to'. The recognition algorithm involves sequential non-deterministic application of rules, with backtracking to test all derivation sequences. (This is claimed as novel.) The editor is syntax-directed for graphics, and free for text (for flexibility); all graphical constructs have an equivalent textual form – diagrams are modelled as directed node and edge labelled graphs with attributes on nodes. Editing modifies the underlying logical document's abstract syntax tree skeleton, which is 'unparsed' to modify all current views of the document. The editor code is generated by their IPSEN meta-environment, from EBNF specification plus text and graphic unparsing annotations.

Schürr *et al.* (1995b) describe PROGRES *Graph Grammar Engineering* as aiming to establish a new specification and programming paradigm. They conclude that the approach is not restricted to the abstract syntax graphs used in CASE tools, but can develop very general complex data structures. In order to develop large systems, they require that efficiency of graph rewriting must be improved, flat graphs must be replaced by hierarchical graphs (with inter-graph edges), and a module concept introduced. The formal semantics is given in (Schürr 1994c).

GRAS (Kiesel *et al.* 1995) provides basic operations such as creation / deletion of nodes and edges, manipulation and incremental computation of attributes, according to a graph scheme defined in PROGRES. The paper describes techniques used to promote efficiency, such as attribute dependency graphs (for lazy evaluation) and the clustering of stored data according to usage.

Structures are described by graph schemes (notated textually, but with an equivalent entity-relation diagram form). Graphs are labelled digraphs with attributed nodes that denote objects; each node has a type, and each node-type belongs to a *node class*. Edges denote binary relations, without attributes. Edge types represent *intrinsic* relations between nodes of certain classes or types. Paths (specified by path expressions) represent derived relations that are calculated from edges and node properties.

Rekers & Schürr (1995b) present a graphical parser that supports free editing of drawn diagrams,

to be implemented as part of the PROGRES environment. It uses a directed graph model. There are four stages: Pictorial elements of a drawing are fed to a Graphical Scanner that yields a spatial relations graph; Low-Level Parsing deduces an abstract relations graph; High-Level Parsing uses a grammar to create a syntax derivation graph. A proof of correctness is to be found in (Rekers & Schürr 1995a).

The output of the process may be a yes/no reply or a sequence of production instances, or a full derivation graph with all non-terminals — or the YACC approach of attaching an action to every production, yielding an action sequence that can generate a data-structure.

Efficiency is achieved by recording the dependencies of all candidate rewritings as *above* and *exclude* relations – analysed from bottom-up. Starting from an empty graph, the top-down phase builds a derivation from the dependencies. Each rule-rhs must be connected, and is equipped with a *search plan* that determines the order in which the match must be constructed. The grammar must be acyclic to avoid non-termination. Ambiguity is admitted when distinct derivations are found, but distinct derivations may sometimes be equivalent, leading to duplication of effort. Rekers & Schürr propose *history relations* to avoid this problem, improving upon Marriott (1994) and Golin (1991b) who use *cover checks*, which restrict the context elements to be terminals.

3.3.3.3 Escalante

McWhirter's Thesis (1995) addresses some generic problems of tool support for general visual languages, and describes the solutions embodied in a system called Escalante. He notes that interface development environments have provided very little support for defining the application model and its representations, and interaction tasks. His modelling places primary emphasis on the meaning of a visual language, with mode of representation a secondary concern. Escalante contains a language specification environment (*GrandView*) that supports refinement and generation. *Grand* is itself a visual language. An object-oriented approach allows structuring of both the external representation and the internal semantics of the visual language. Automatic graph layout is not covered.

McWhirter claims that the system enables applications to be developed in days or hours, and that it can support a much wider range of visual languages than the other systems he reviews. To do this it uses a *characterization framework* that serves as a conceptual meta-language for 'the

underlying structure' of graph-based languages. The graph characterization is used as a *lingua franca* for constructing applications. Examples given are Petri Nets, Bar Charts, Tables. The framework is intended to be powerful in its scope, in order to describe abstract relations that are implicit in diagram structure; it is not intended for, and does not easily apply to, complex fine grained geographical information.

> A visual expression is formulated as a set E of attribute-tuples, partitioned into binary relations R and entities N. The attributes are those required to specify syntactic and semantics. The framework addresses the propagation of changes to attribute values. Each tuple has a *type* (e.g. circle, arrow) based on its sequence of attribute tags. Relations are the domain of [polymorphic] functions *head* and *tail*, which take values in E, thus giving a higher order relational structure [not just a directed graph].

Behaviour (operational semantics) is addressed by a set of mechanisms that act on the graph constructs to define a subset of their syntax and semantics. An event (e.g. during editing) is treated as an operator; it is applied to an element, propagated to the incident relations and connected elements of the element, by means of a specified set of *event maps* associated with a relation. Cyclic propagation is disallowed. An event may not change the structure of the graph – deletions are carried out by marking followed by a global clean-up. An attribute propagation mechanism is specified by a set of *attribute maps* associated with a relation. This includes a filter function that defines changes to values, and a constraint function that constrains values.

3.3.3.4 Some Others

McWhirter (1995) reviews *GLIDE* (Kleyn & Browne 1993). GLIDE is a formal language for describing graph based languages and environments, by means of a structure grammar, view queries and transition predicates. The latter are of three types, for editing, execution (for dynamic semantics) and animation (for mappings between the state of a language element and its graphical representation). The proposed usage is to compile a language specification onto a pre-existing visual language substrate such as EDGE.

Jorge & Glinert (1995), whose use of adjacency grammars is described above (see §3.1.5), have produced a visual compiler-compiler that generates C++ code.

3.3.3.5 Adequacy of Notation Tools

It is evident that a considerable amount of research is being pursued into development of tools which can process graphical notation. The reports here show that much headway has been made

in recent years, in practical terms. It is only appropriate here to criticize these tools in respect of the way they deal with notations, rather than their full function. Just a few tools come near to being satisfactory as generic notation editors, in that they have sufficient generality of application and are based on formal specification of notations.

The most interesting of these tools is DiaGen, because of its claimed versatility. The hypergraph formalism employed is a close generalization of textual grammars. PROGRES is important because of its breadth of scope in kinds of processing and its methods to improve efficiency of parsing. The uniform use of graph grammar techniques is combined with the separation of spatial relations and abstract syntax as proposed by Rekers (1994). The approach of Üsküdarli & Dinesh also provides formalisms for both graphical and abstract syntax, but without a common basis, and the term rewriting approach seems too restrictive by comparison with graphs. Escalante is a good example of an approach that uses a complex ad-hoc formalism in order to satisfy computational goals, but without justifying the power of the techniques in terms of notation properties. In common with other VPEs, there is no separation between formal semantics of the language and operational behaviour of expressions; no meaning can be attributed to an expression without executing it in a context.

Several methods use variants of Prolog to encode grammatical representations and parsing procedures. This makes the specification of a notation into a programming exercise; the user must be able to predict the interactions between rules. Prolog is attractive over non-logical procedural languages, because it hides many implementation decisions relating to searches. Prolog is an executable formalism, however, and not purely a declarative language.

Describing syntax is intrinsically simpler if the user is only required to declare properties, and not plan the execution of parsing. Checking for executability is then the job of a tool for constructing editors from syntax specifications. When a specification is compiled to generate a parser, the compiler must test whether it can be implemented efficiently.

3.4 Discussion of Problems and Issues

Following the survey on notation in the previous chapter and the reviews of techniques, formalisms and processing tools in this chapter, we are now in a position to highlight the issues raised and to summarize the strengths and weaknesses in the approaches covered. The purpose

of this section is to bring out the main challenges that must be met by research efforts. The discussion is divided into three areas: notation design, mathematical description and tool support. For each topic in an area, important points are organized into a numbered list and explained in a short commentary.

3.4.1 Problems of Notation Design

Based on the literature surveyed in Chapter 2, a summary can be presented of notational needs in a software development context. From the analysis (§2.3.1), several points of difficulty are to be found in the requirements that notations must satisfy. Further points of difficulty describe the current lack of knowledge of practical methods for design of notations.

3.4.1.1 Difficult Conditions and Demands

N1 Coping with the complexity and size of software systems is problematic.

N2 Software is non-material and hard to understand without using representations.

N3 Notations must cope with a wide range of kinds of system, and express different aspects of behaviour.

N4 Notations are used in different ways that demand conflicting features.

N5 The need for rigour increasingly places *formal* demands upon notation.

N6 The need for reasoning and computation can lead to technical formality that conflicts both with ease of use and flexibility.

N7 All aspects of notation use need to be supported by computer aids.

The survey of Chapter 2 indicates two main reasons why software development needs notations, and why the needs are difficult to satisfy. Firstly, software systems are often large and complex objects [N1]. With large systems, notation can depict an abstract analysis into a hierarchy of named units. As a result, the notation may then lose some directness of expression, making reasoning less easy. Secondly, unlike many other artefacts, software is not directly appreciated by the senses [N2]; design must therefore go through several stages of abstract graphical representation before models and prototypes can be built. These stages require notations that can fulfil the many differing roles found within the development process.

The wide variety of purposes [N3] warrants using a whole system of notations in many styles, suited to different participants, different methods and different application areas [N4]. The style

tends to be diagrammatic and informal in the early stages of requirements analysis and design of overall system structure, but textual and formal in later stages that produce program code. The supposed advantages of pictures over text conflict with the need for rigour and formality in notation. Diagrams, for instance, are rarely logically expressive enough for general use in specifying requirements, and they are not sufficiently concise for recording a mass of detail.

With the introduction of formal specification and refinement techniques [N5], we find that notations are increasingly subject to *formal* demands, owing to the need for accuracy and reasoning, and the re-use of abstract structures. The practise of Formal Methods, for instance, imposes a formulaic style of logical expression in all stages. Since expression is generally not just a personal matter to assist individual problem-solving, documents must be precise and clear, and their meaning widely accessible to technical personnel. Program code and logical formulae are precise and formal [N6], but adhere to an inflexible, restricted syntax. In the practice of reasoning and calculation, a more flexible style of expression is normally preferred, owing to the confidence that results from the presence of a formal semantic basis. Such universal standard forms of notation require a firm logical basis so that their style can become less strict.

Some authors see graphical notations as *intrinsically* desirable. Although textual notation may be preferred for formal work because of conciseness and ease of manipulation, patterns of reasoning may be better motivated by graphical analogy. For accurate calculation, paper and pencil give way to computer assistance [N7]; there is a similar need for flexible computational support in working with diagrams, and all styles must somehow be formally related.

> What determines choices of style in notation design? Many formulaic notations, such as the logic languages of mathematics, attempt to accommodate varied semantic constructs in a common graphical and syntactic format. This makes it possible to express systems of arbitrary structure and complexity in a simple manner.[5] It can also be desirable, though, to link syntactic form closely to structure represented – even though this does tie the complexity of an expression to the complexity of the subsystem being expressed. This alliance between syntactic form and semantics is an aid to reasoning, as reported above (§2.1.3); diagrams commonly depict directly the operational units (functions, modules, objects, procedures or whatever) that are found in a software system under consideration.

Thus practical concerns in software development raise issues of notation design and how it is

---

[5] As Stenning & Tobin (1994) put it: " In text, a single representing relation (concatenation) is heterogeneously semantically interpreted".

supported.

3.4.1.2 Difficulties in Design Methods for Notation

D1 There are no established principles to support design of notation, and little formal attention has been given to diagrammatic design.

D2 There is little discussion of the way notations carry meaning; the structure and function of metaphor is poorly understood.

D3 Central to the activity of notation lie some fundamental logical issues.

D4 Text and graphics are treated as opposing alternatives of expressive technique.

D5 It is hard to design formal notations to support informal discussions.

The demands on notations are hard to fulfil [D1] because diagramming technique has evolved in an haphazard manner and there is a lack of design science for notation to help in these application areas. In the main, notational choices are not designed; they are arbitrarily adapted from past practice. Rarely are notation designs justified in the literature except by personal experience of their use. Although mathematics has been applied to specific cases (usually to explain semantics) in a piecemeal way, no coherent body of "notative concepts" or structures has been recognized as deserving study.

The starting point for design is the requirement to carry meaning and aid reasoning. Meaning and denotation are, however, rarely formally defined [D2]; outside of textual programming languages, denotational semantics is not used. In order to assist reasoning, it is essential to incorporate spatial, kinematic and other metaphors into the graphical design of diagramming. Though analogy has been studied within some important investigations into diagrammatic reasoning, software engineering notations have not yet been treated. The design of diagram syntax needs to start from the logical structure of the subject domain, which must be explained or depicted by analogical means. A formal notation of any kind then embodies a logical system [D3], which may not have been fully studied either by the notation designer or elsewhere.

Once logical issues are settled, design can attend to the more concrete aspects of syntax. Notation design needs to exploit the different advantages of both text and graphics [D4], combining formulaic and diagrammatic features to best effect. For flexibility, there must be an adjustable balance between abstraction and specificity (directness), allowing choice of which configurations are abstracted. The hiding of detail made possible by abstraction opens up a way

to offer a more vague mode of expression [D5], which may be appropriate at the early stages of a software design process, when ideas are often diffusely represented. The preciseness of notation cannot be controlled until the concepts of ambiguity and vagueness are clearly defined.

3.4.2 Problems in Specifying Notations

Next we consider how notation is specified, taking note of several issues and openings for research, and highlighting some points on the lack of clarity over the subject of study (artificial notations), the problems of description and the choice of suitable formalisms.

3.4.2.1 General Issues of Specification

S1 The techniques and tools present us with a jigsaw puzzle of varied and disparate approaches.

S2 There are no theories that define what constitutes an artificial notation, as opposed to a system of representation, or a spoken language.

S3 Informal definition of syntax may obscure the underlying properties of the subject domain.

S4 Mathematical formalizations help the specialist, but cannot be controlled by most users.

S5 Disagreement exists on the underlying type of logical structure presumed in syntax.

S6 Spatial relations and theories are important, especially in regard to analogy.

S7 Descriptions of syntax involve a mix of logical constraints and structural rewrite-rules.

S8 An *operational* semantics for a notation can be given only if the context is formalized.

We would hope for an uniform theory on which to base descriptions, but this is not what we find. The pieces of the jisaw [S1] – qualitative spatial logic, grammars, algebraic semantics, declarative programming systems, analogical inference, graph rewriting and constraint logic – must somehow fit together to form a coherent picture. This predicament is acknowledged in recent work. Marriott & Meyer (1996) observe that progress in visual language specification has resulted in a wide range of formalisms that are hard to compare owing to diversity in underlying assumptions. They conclude that a common basis for specification is indispensible to gain clarity. Haarslev (1995) observes that there is "still a strong need for an adequate theoretical foundation of visual languages." On the evidence collected in this chapter, a convincing common basis is yet to emerge.

Another problem is the uncertainty of scope. We do not find clear definitions of what counts as a

notation or visual language [S2]. This causes difficulties, because the limits of a descriptive method must be determined from the range of phenomena to be described. Approaches differ in their treatment of syntax and semantics; for example, many notations we wish to support do not have the clear operational semantics required in the specialized area of visual programming. We must allow that the reported formal notions of graphical notation may not be defining the same thing – the pieces of the jigsaw may not all belong to the same puzzle.

Many aspects of representation in computing are not notational. For instance, data and program structures are not in themselves notations, though they may be visualized. In this thesis we require notated expressions to be designed for people to view or read, but it is unclear what is in common between graphical language and spoken language. The linguistic theories in (§3.1.1) show that natural language is complex and rich in structure. For diagrams, it may not be possible to formalize all the notative mechanisms which may arise or evolve naturally, owing to the indefinite number of pictorial metaphors that could be created. If we restrict our aspirations to notations that are artifacts, formally devised for particular purposes, some simplification is essential. To base an analysis of notation structure on theories of natural language would build in unjustified complexity. Neither can it be assumed that graphical language adopts the same structural tactics as are found in spoken language.

The thesis argues against the view that graphical notations should only be used as an informal aid in the development task, or in explanations to users. Diagrams with no precise meaning cannot be acceptable for accurate communication with users. There is then a need for appropriate methods of definition. If syntax definition is informal [S3], the true behaviour or semantics of expressions may differ from the user's intuitive understanding of the analogies embodied in the notation. This informality cannot assist a rigorous approach to software development. Mathematical techniques can support syntax definition [S4], but are also hard for most users to read and comprehend. Specialized methods each rely on a particular notion of *structure*. Research must establish what kinds of structure are suitable.

The structure of expressions [S5] underlies any definition of syntax; it is a data-structure, normally some kind of graph or tree, upon which computations can be performed. Such structure is not essentially *spatial* – to become so it requires the help of an analogy [S6]. Much of the recent work has emphasized the spatial properties of diagrams, in terms of *qualitative* topology rather than the

mathematician's usual formulations. Suitable 'theories of place' are needed if software is to recognize hand-drawn forms or analyse perception; their relevance to syntax and semantics is more a matter of defining the analogical relationships between the data-structure and pictorial structure.

Several different kinds of rules are proposed for syntax and semantics. Syntactic rules [S7] can be grouped into those that *generate* expressions and those that *constrain* them to be well-formed; constraints may apply either to the results or to the process of rewriting. Both grammatical parsing and constraint-checking rely on pattern-matching algorithms, but the connection between these two kinds of rule is not well understood. There is thus a spectrum of specification techniques that ranges from wholly constraint-based techniques to those that use only rewriting. At one end of the range, computation relies on constraint-solving; at the other, defined structure is specified as that which can be generated by some grammar. The techniques vary in how much information on implementation is supplied in specifications. For semantics, some *operational* approaches [S8] are applied in the case of visual programming environments (§3.3.3). The notation processing requirements reported above (§3.3.1) suggest that operational rules may be usefully formulated wherever the usage of expressions within a context is sufficiently well-defined.

3.4.2.2 Problems of Syntax Description Formalisms

F1 Specification techniques are too powerful and general.

F2 Formalisms do not embody specific theories of notation syntax (see S2 above).

F3 Formalisms do not clearly separate different kinds of structure.

F4 The grounds for using graph grammars to describe syntax are not clear.

F5 There are no clear guidelines for determining which type of graph to use (see S5 above).

F6 Implementation of general rewriting rules is a non-trivial task, and parsing is complex.

F7 Graph grammars do not easily treat spatial reasoning and global constraints.

F8 In most techniques, specifications for graphical notations are expressed in textual format.

From the varied methods reviewed in this chapter, we can infer that necessary flexibility is only to be achieved by a highly abstract approach to structural representation. Where the difficulties lie is in how the expressive power of formalisms is managed [F1]. Greater power has been introduced in order to ensure coverage of a wide enough range of notation structures. Programming languages, logical languages and graph grammars are technical tools that have great generality of

application; it is thus not surprizing that they are useful for defining various diagramming methods. It is certainly valuable to specify notations using a specification language such as *Z*, but such general purpose formalisms fail to shed light on the simplicity of notation structure.

Attempts to devise formalisms especially for specifying notations have not yet uncovered the appropriate limits on expressiveness; properties peculiar to notations have not been taken into account [F2]. If the general principles of graphical mechanisms were better understood it would be clearer how to provide a sufficiently simple theory – rather than rely on general methods of specification that have been developed in relation to computation. Formalisms therefore need to be adapted to the specific area of notation design.

Methods of formalization address different aspects or kinds of notation structure. A comprehensive method [F3] must be capable of separating different structural processes – whether graphical, syntactic, semantic or pragmatic – which may be present. It is harder to design or modify a notation if the formalization allows boundaries between layers of structure to be unclear. Some recent researches separate pictorial structure from syntax – e.g. VODL/VSDF (§3.2.1) and PROGRES (§3.3.3).

Grammar formalisms are popular; these approaches use graph theory or relational structures to generalize the better known grammars of artificial textual languages, which in turn originate as simplified methods from linguistic theory. We need evidence [F4] that such grammars are in fact relevant to graphical notation. The grounds for choice of graph-type [F5] need to be explained; in the reviewed approaches, we find a full range from simple directed graphs to higher-order relational structures.

Having fixed and formally specified the type of graph, in grammar formalisms the checking of syntactic correctness requires a complex search [F6]. The complexity of computation is too great unless the expressive power of the formalism (and the range of notations covered) is restricted. Reports indicate that incremental graphical parsers are now capable of efficient operation for suitably restricted grammars.

Spatial concepts [F7] are usually dealt with by means of attributes on elements, and constraints on these attributes during rewriting. The logical complexity of these constraints is then an issue. The use of attributes in grammars allows local graphical constraints to be accommodated, but does not

easily manage global aspects of style.

Although many authors emphasize the benefits of graphical expression for comprehension [F8], these benefits are often forgotten when it comes to the specification formalism itself. Whether based on grammars or logical constraints, formal approaches rarely express their rules graphically; this may be because of a presumption that a professional programmer or other specialist will be the end-user. The approaches concentrate on facilitating the programmer's job (of building an editor, say). Certain grammar formalisms do have the advantage of being able to express rules graphically, though they borrow some pictorial items and spatial relations from the notation itself.

3.4.3 Limitations of Notation Processing Tools

We have seen that formal definitions of notations are a prerequisite in constructing tools to assist processing. Taking into account the scope of this chapter, we here consider problems of processing expressions, in particular the deficiencies of graphical editing tools that other researchers have described.

3.4.3.1 The Scope of the Review

The review presented in this chapter does not claim to cover every kind of processing support. Individual computer-aided software engineering (CASE) tools that support notations as an integral feature of a development method have not been reviewed. We have not considered how well CASE tools are able to extract and present requested information in notation suited to the user. Nor have we covered systems that assist human reasoning with diagrams or those that undertake general symbolic computation, calculating or querying knowledge-bases. These are regarded more as problems with visualization of data or computation, going beyond the notational issues that are the focus of this thesis. For our purposes, notation processing does not concern symbolic structures that are too large for display or coherent perception.

The review has not covered the full range of activities that may be supported by tools. Less attention has been devoted to certain semi-automated operations that require little human intervention:

checking pragmatics: consistency with context,
interpreting (especially the immediate interpretation that is needed in a graphical dialogue),
translating and changing of viewpoint (hiding detail, revealing consequences),
automatic or assisted layout control.

The tools considered are predominantly those that support operations that help people in the task of producing expressions:

creating (drawing and composing),
generating (by grammatical rule),
editing (with syntactic & semantic checks).

In addressing needs for notations, we seek tools which can provide help with their specification and design. Generic editors are the only available systems that might serve this purpose.

3.4.3.2 Problems and Limitations of Processing Tools

T1 Building of notation-interfaces for software engineering tools has resulted in fragmented effort.

T2 The lack of coherent notational design principles makes tools unsuitable for developing new notations. (see D1 above)

T3 Current generic tools are limited in the features and structures which they can accept in a newly designed notation.

T4 The notation-user is mostly excluded from the process of shaping notation, with little opportunity to create, amend or reason about the specification.

T5 Current tools are limited in the processing they implement – principally editing and compiling, with little support for translating between notations or offering variant views.

T6 Automatic layout satisfying global stylistic constraints is inherently complex.

Because of the common close association between notations and method, tools have been far below the level of general application necessary for user-control of notation or standard construction. Editors for each notation used have been programmed individually [T1], though this state of affairs is changing.

The reviewed tools do not give sufficiently broad support for processing. Because they do not have appropriate theory to rely on [T2], tools offer little help with syntax design. Each system is developed on its own individual theory and implementation of notation structure. Fully generic notation processing tools [T3] need to be based on a sufficiently general characterization of

semiotic structure, so that innovations in semantics or syntax do not force *ad hoc* extensions to be made to the systems.

Even the use of improved techniques such as graph grammars or constraint logic [T4] does not offer users an accessible way of modifying syntax. As a consequence of the over-complexity of specification formalisms and lack of software support for manipulating them, it is difficult for users to alter the specification of a notation to suit special purposes. Recently developed systems still require expertise in logic and programming in order to design or modify notations. A coherent approach to notation design needs to be more than a graph grammar / constraint logic programming exercise.

Tasks other than editing [T5] are neglected. Semantics of programming languages is only supported in a concrete operational sense by interpreters and compilers; neither formal denotations nor translation to other languages are normally supported. Expression layout [T6] may require a range of techniques that do not fit into grammar models.

3.4.3.3 Difficulties of Editing and Editors

E1 Editors cannot easily offer varying degrees of guidance suited to the individual user.

E2 Editing an expression may involve complex operations not defined by grammar rules.

E3 It is hard for grammar-based editing to accommodate sensitivity to semantics or context.

E4 Unfinished expressions which are generated during 'permissive' editing are not easy to store.

E5 There is a lack of awareness that different *depths* of structure-checking are needed.

The need for editing with enough freedom and flexible guidance [E1] is hard to satisfy. Guidance should ideally range between recognising free hand drawn input and demanding fully syntax-directed selection. Which rules should restrain manipulations? Which rules should cause warnings before a requested change is confirmed? Which rules should be applied only as checks, on request?

Guidance for editing must be derived from the syntactic specification. If syntax is defined by constraints, checking for well-formedness involves logical inference. If syntax is defined by a grammar, checking requires a search for derivations. It is not clear whether either method has a general advantage. When using a grammar, the operational nature of rewrite rules has an affinity

with certain editing processes, but editing is not simply a matter of applying compound rewrite rules, as some tools presume. Composing an expression [E2] may involve operations of splitting, joining or substituting of expressions.

During the creating of an expression, the context in which it is enacted (e.g. integrated with a larger document) may not yet be determined. If the context of enacting the expression is known during editing, sensitivity to semantic and pragmatic compatibility becomes possible [E3], but only by logical inference or computation.

Unfinished expressions [E4] often have no clear status; they are not made available as legitimate vague or partial representations. To allow for partial expressions, checking would need to be selective. Checking could be carried out according to which level of constraint [E5] the user wants applied (graphic, syntactic, semantic, pragmatic, stylistic). The computational models underlying editing should not interfere with these requirements for flexibility.

3.4.4 Researching Notation in Software Development

In view of the problems just discussed, and in order to clarify the choice of topics for research, we next consider some of the benefits that may result from formalization. We look at the assistance that it offers in the difficult areas of designing, utilizing and processing of notation. Possible approaches are suggested to solving the above listed problems.

From the many detailed points noted in this section, two general points on formalization stand out:-

W1 The weaknesses of current approaches stem from a scarceness of clear theory that is appropriate to notation specifically.

W2 The various techniques and tools are too diverse or rely on unclear principles; they offer users little flexibility, and do not readily and reliably extend to new notations.

This thesis offers a more uniform formalization as a way to address the problems.

3.4.4.1 How Formalizing Graphical Notations can Help

H1 Provision of a theoretical basis for graphic notation can lead to greater flexibility and expressiveness, and an increased confidence in precision.

H2 An uniform specifying formalism can allow notations to be compared in structure and complexity.

H3 A formal description can make room for informal variations and annotations.

H4 Formalization assists development of both mathematical logic *and* pictorial metaphors.

H5 Rules of manipulation are established by formalizing the notation.

H6 Formalization can provide modular specifications to help make designs flexible.

H7 The programming of graphical notation tools is reported to be much simpler using specialized formalisms.

H8 Graph grammars and constraints are each useful for controlling the operations of editing.

H9 Translation between notations from different methods requires some common structural basis.

In contrast to an informal approach, a formalized basis [H1] opens the way for logical rigour in all styles of expression. Formalization provides an uniform framework [H2] capable of specifying the different aspects of structure that can be found in notations, and which may be used to control the complexity of syntax. Within a formal framework, aesthetic or informal details of notation style [H3] can be accommodated by heuristic rather than exact rules.

Formalization applies especially to purposes of reasoning and calculation [H4], where concise algebraic formulae are especially valued. Although the structure of such formulae is well understood, their semantics cannot easily be directly expressed in diagrammatic style. A formal understanding of semantic processes allows better use of metaphor and analogy, that is necessary in converting to formulae into diagrams.

Tasks which insist upon rigour [H5] may be made easier to grasp by the use of diagrams and computer-aided manipulation. Formalization admits automatic application of rules and assisted heuristic searches for solutions, although reasoning cannot be fully automated. Notations can then be given an instructive semantics which helps users to think and calculate, through applying explicit rules.

Separating different kinds of structure [H6] within a notation helps support flexibility of design – users can then change the more superficial syntactic characteristics of a notation without disturbing the semantics. There is a great need to allow variation of modality and all aspects of style. For example, the pictorial elements in a notation can act as metaphors, by suggesting some intended analogy. It is therefore helpful to allow the shape or spatial relations to be changed in order to select the best metaphor. Separating different kinds of structure within a notation also helps keep specifications simple and in principle allows more efficient implementation.

Formal specifications are essential in supporting editing [H7]. Editing involves modifying a drawing, subject to constraints that define the notation. The specified syntax in effect determines the maximum structural constraint that can be applied during the task of composing an expression – outside of context. Several reports of recent research show that progress is being made on more general and flexible support for building notation editors from specifications. The particular method of specification [H8] can affect the process of editing. Broadly speaking, graph grammars are suited to directed editing, but constraints are better for permissive editing. Improved techniques of graph rewriting [H9] also make it possible to use graphs as a *lingua franca* for translation.

3.4.4.2 Potential Methods of Implementing Notations

M1 Logic and logic-based computing are valuable.

M2 Graph rewriting techniques are important.

M3 A hierarchy of expressive power in formalisms is valuable.

M4 Processing should take advantage of spatial layout to optimize searches.

The reports show where solutions may fruitfully be sought. The methods of specification that have found most favour are similar to those being applied in computational linguistics [M1], which are allied to declarative programming, logic and type theory. Yet there still remains a theoretical gap between generative grammars and logical constraints. some sort of constructive or operational theory must govern the manipulation of notational structure in editing and translating. However the structure of expressions is framed [M2], rewrite rules may provide an apt formulation for operations of transforming and calculating, independent of programming languages.

Efficiency should be promoted by graded complexity [M3], based on different kinds of syntactic structure. If this is not done, processing and reasoning will be no more tractable than in operating on general structures. Efficiency can be aided by the two dimensional layout [M4] that is a defining feature of notations. Just as the concatenated structure of text or spoken language is central to efficient parsing, in diagrams the 2D spatial structure can be used to help organize efficient searches during structural matching.

3.5 Selecting the Research Agenda

Now that we have observed where the difficulties and challenges lie, and given the confines of a doctoral thesis, the task remains to make a selection of research goals that may be achievable. The choice is guided by the issues that have emerged in the primary literature that is reviewed above. In the course of the present work, the importance of these issues has been confirmed by the more recent of the reviews.

A way forward is proposed, and the aims for this research into formalizing graphical notation are established. Finally, a list is given of some topics that must be excluded from our consideration in this thesis.

3.5.1 A Proposed Way Forward

As indicated in the above discussion, this work contends that formalization is a key to solving the problems of graphical notation. Here the reasons are summarized and a statement of aims and objectives is presented.

3.5.1.1 Formalizing Graphical Notations

Diagrammatic expression has a folklore of practical techniques and conventions, but until recently there has been little theory to explain why these notations work or even to describe them. By *formally* specifying the structure of pictorial notations, the ground may be prepared for establishing good design principles. A suitable research aim should address this task with an eye to constructing and applying as simple and appropriate a body of theory as may be found.

Research should establish a uniform basis for notational design – one which takes into account semiotic principles, and which yields elegant ways of combining constraint logic with graph grammar techniques. The basis should be able to explain metaphors. Formalizations of notation structure that stand on this uniform base should then be expressed graphically, in several specially tailored specification meta-notations. This would make semiotic structure more explicit and easier to understand, thereby increasing awareness of design choices.

The benefit in formalization lies in helping to improve the design of notation and to give practitioners – users of notation – more control. One way to do this is to provide generic notational-design tools, which could be used whenever a need for new notation arose. These

software tools would enable standard editors and semantic checkers to be built, without the need to use a programmer's language. Although inventing a notation 'from scratch' would still need special skill, the facility to make minor adjustments and extensions to existing notations would be within reach of many users.

Computer-aided software development requires a system of notations for different purposes. Ideally, a notational-design tool would provide a facility for permitting incremental change to syntax and semantics for all the notations designed or adapted for use in the developer's method.

3.5.1.2 Statement of Aims and Objectives

In accordance with this analysis of the problems, this thesis aims to put forward a mathematical theory of semiotics for notation systems in order to describe notative techniques more formally. It will apply mathematics to the problems of designing effective notations, and of building interactive tools for notation processing.

The intention is to lay down some stepping-stones towards a science of formal notations.

The objectives of the research are to find:-

1) a formal, uniform means of specifying the structure of graphical notation systems;
2) a computational and mathematical foundation for designing graphical notations;
3) a clear diagrammatic way to communicate syntax;
4) a plan for developing a generic notation-processing tool, with a prototype implemented in Smalltalk.

As this chapter has shown, during the period covered by this research many other researchers have begun to tackle related problems concerning visual language and diagrammatic reasoning. The lessons and gaps discernible (§3.4) in these parallel researches will be given further attention in succeeding chapters.

3.5.2 Excluded Topics

There are many interesting topics related to this study that will not be covered here – they are addressed by other authors.

3.5.2.1 New Modes of Expression

The computer is also a new medium with extra dimensions of representation (Colour, 3D effects,

Animation and Interaction) that increase possibilities for analogy. Software writing has already been simplified by the advent of *Visual Programming*, reducing the need for coding skills. *Program Visualization*, which seeks to represent the execution of a program, must rely on interaction with animated graphics. These aspects will not be treated in this thesis.

3.5.2.2 Interaction and Dialogue

A software development method makes use of a system of several notations and a project will generate documentation expressed in them. The documents "tell a story" of the development, which is open ended, always subject to modification. An exchange of diagrams can take place between person and software, as in 'query-response' with a database. The effect of *enacting* an expression in a document is to modify the context in some way – perhaps adding to a specification, or giving an instruction to a software application, or providing data to an active process. In return, a process might produce an expression, as if in dialogue or *discourse*.

Another kind of discourse occurs in 'direct manipulation', where an expression on screen becomes a communication channel – a part of the context. Using gesture as a communicative act, a person creates a signal, to which the computer may respond by changing the expression. Interactive notation, by means of pointing, pressing and dragging, exploits *haptic* senses and follows kinematic and mechanical metaphors, that are neither linguistic nor visual. This may be ascribed to a *dynamic* syntax of interaction, that extends graphical syntax into the gestural medium of *user interfaces*. Study of these kinds of discourse will not be the focus of this work.

3.5.2.3 Cognitive Principles

Formality clarifies the details necessary to support computer-aided editing, interpretation and translation. Clarity of design may also lead to a better understanding of human factors, of how skills place limits on the size, detail and style of diagrams, depending on the context in which people meet with the expressions. The principles that make a notation easy to learn, or improve legibility of expressions, must however be informed by studies of cognitive and perceptual ability, which are outside the scope of this work.

Chapter 4

An Exploration in Search of Notational Theory

Abstract

Here we find an exploration into the nature of graphical notations and possible formalizations, which seeks to resolve the problems noted in the previous chapter. First there is a clarification of boundaries for the topic of research, and an analysis of the roles that technical notations fulfil. The exploration then ventures into elementary semiotic concepts, discussing these as they relate to notations. The discussion argues that notation draws upon many prior cognitive skills in order to motivate its connexions between signifier and signified, whether linguistic, pictorial or spatial. This posits iconism, analogy and metaphor as initiating principles for signification, though the association of meanings can only be *established* by usage and agreement. A phenomenon of layering is noted in general codes, which may be attributed to economy in cognitive specialization.

Ideas of computational linguistics are explored next, with reference to our main concern of defining notation structure. These ideas suggest that the logical relations between concept and percept are organized to make deduction of meaning feasible, and that the grammar rules act as a resource-sensitive deductive system. The question of structure is resolved by taking the form of expressions to be a certain 'graphoid' structure, in order to support rewriting and local computations.

These arguments lead towards a theoretical framework; according to this proposal, notation is described by a formal theory divided into layers, with mappings between theories to define semiotic process. An expression is then a model of a syntactic theory, and grammars arise as implementations of proof-strategies. A continuum of inference connecting graphical elements to semantic concepts explains how both learned rules of manipulation and analogical mechanisms help the viewer of expressions to concretely verify their thinking.

Thus the chapter points the way towards a formal understanding of notations as sign-systems, and lays a foundation for an uniform descriptive theory in accordance with the aim stated in the previous chapter.

Chapter 4.

An Exploration in Search of Notational Theory

"... the purpose of theory is to organize thought, not to drown it, to be constructive without being oracular." – Vaughan Pratt (1988)

The survey and review have directed our attention towards a general area of difficulty in describing and processing the notations needed for software design – an area in which better methods of formalization may be of some help. The purpose of this chapter is to investigate the nature of graphical notations and to look for appropriate ways of formalizing them and addressing the problems raised.

The chapter determines more carefully the area that is to be researched and situates it within the wider territory of semiotics; an analysis of semiotic concepts is undertaken to shed light on the mechanisms and structures found in various styles of notation. The central issue, of notation structure, is then explored in some detail, informed by the observations of computational linguistics. In the final section the argument leads towards a proposal for defining notational processes, thereby laying the foundations for a theoretical framework.

4.1 Defining the Area of Research

Our exploration starts with a consideration of the proper ground for research into formalizing notation. This section discusses the nature and origins of notation, continues with the notion of formality and then analyses in more detail the kinds of roles that notation plays in a technical environment. This will help us focus more clearly on specific problems of reaching a formal understanding of structure.

4.1.1 The Nature of Graphical Notations

To clarify the focus of study it is necessary to set some boundaries around the topic. We turn to the early history for clues about the commonly observed properties of graphical notations. We reflect on how notations differ from languages, in usage and attributes.

4.1.1.1 What is a Graphical Notation?

Graphical notations are limited, for the purposes of this thesis, to systematic formations of

expressions that are *communicative* and can in principle be *drawn* without computer support – ruling out spoken language, data-structures, images and patterns as such. Just as structural linguistic theory excludes acoustics and cultural knowledge from its remit, an examination of notations should also stop short of analysing graphical displays and computer science.

Signification in notation is therefore based on codes that are constrained by human abilities to apprehend drawn configurations. Although often referred to as *visual*, there is no restriction to sight as the mode of perception; *tactile* sensing could also stimulate the relevant spatial cognition, along with linguistic and more general pattern comprehension – as noted in Stenning & Oberlander (1992).

4.1.1.2 Text and Diagrams

As we have seen in (§2.1.3), diagrams are generally regarded as very different from texts. What is the basis for the difference? For instance, (Gurr 1996) states:

> "The two most notable differences between texts and diagrams are the relative difficulty of expressing abstraction in diagrams and the inherently one-dimensional nature of texts."

He relates this to the observation that textual representations are *type*-referential (identical tokens refer to the same object), while diagrams tend to be *token*-referential (alike tokens refer to different objects). Barwise & Etchemendy (1995) point out that good diagrammatic representations always exploit features of the domain being represented, and so typically lack the representational expressiveness of language.

Nevertheless, diagrams are not fundamentally different from text. Written words and pictures are both signs within systems; they have a common communicative purpose amongst those who use them. This thesis argues against the taking of too absolute a division between verbal and pictorial modalities, based on cognition. Sequential form is one of many possible graphical arrangements, and abstraction is a principle equally available to diagrams – either in the sense of reducing unwanted detail or of recursive coding. In placing the focus on *notations*, this work seeks to include both modalities on an equal footing.

This thesis prefers the neutral terms 'notations' and their '*expressions*', acknowledging that diagrams and text are both *graphical* – drawn, stylized forms. A more important distinction to observe is that between the persistent physical status of expressions and the transient nature of

the spoken word, which makes different demands on mental mechanisms. Many *cognitive* mechanisms are surely at work in grammatical processes and in linguistic or pictorial metaphor. Although cognitive constraints must be taken into account and are worthy of study, this research primarily treats notations as *'cultural'* entities.

4.1.1.3 Origins and Usage

In what ways do human cultures determine the form of notations as distinct from languages? Why do technical notations appear in two kinds, diagrams and formulae? Knowledge of the origins of notation would go some way towards answering such questions. It is not possible here to establish how notations originated, but we can at least consider how and for what purposes they have been used.

Archaeological evidence suggests that the ability to draw pictures made possible the development of script styles from the spoken language codes that long preceded them. These *written* symbols, whether they had phonetic value or semantic sense, soon became stylized and detached from their earliest pictorial significance. It appears that writing was for a long time practised only by a minority selected for their particular intellectual skills. Although these skills are today more generally attained, writing in technical areas is still restricted to minorities with specialist training. In mathematics and the sciences, a *formula* is a shorthand for a sentence, replacing words by symbols, whereas a *diagram* is a stylized drawing. Just as with writing, these *notated* symbols, whether derived from alphabets or from pictograms, have become detached from their earliest context to achieve a newer, more precise meaning.

These origins imply that, as with language, notations are a means of expressing, sharing and recording ideas. Early pursuits of Arithmetic and Geometry show notations specialized for structurally complex ideas that are not easy to capture concisely in speech. They offer precision, since usage is cleaner and less ambiguous than with language. As well as being a shared medium of *communication*, they assist *reasoning* and *calculation*. Diagrams help *visualize* structure, while symbols in formulae can be manipulated by rules. The ability to formulate rules for calculation is quite a recent development of human skill.

The visual presentation, as with writing and pictures, has advantages over speech. Expressions persist, and can hold attention long enough for a story to be read. Viewers can point to them,

4.1.2 Formality and Formalization

We continue by considering what it means to formalize a notation. What are the implications of such a procedure?

4.1.2.1 Formality in Language

In a computing context, formality is usually defined as the use of mathematics, though it is not in fact an essential part of applying mathematical techniques. The need for formality lies rather in the global spread of communication, where sharing of context and experience is limited. This is one reason why programming languages demand high standards of formality. It also explains why normal notational practice in mathematical investigation is informal. Formal work is reserved for checking and verifying calculations and proofs.

An extreme case of a lack of shared experience occurs between human thought and mechanical computation. Formalizing a procedure is essential in order to enable automated calculation and reasoning. Computers can perform symbolic calculations efficiently, using concise programming codes that have low redundancy in an information-theoretic sense, but are therefore prone to error when in human hands.

Informal language relies on the 'common sense' resulting from an ill-defined body of experiences shared in a community. Computers, on the other hand, are less well adapted to informal non-symbolic tasks, such as picture recognition and general problem-solving. They cannot resolve ambiguity in an expression by using unencoded common sense. Hence computing demands more formality in notations.

4.1.2.2 Formalizing Notation Structure

We find two senses of the word *formal*. The first is that of being constrained by some fairly stable or strict cultural code of behaviour. In the second sense, a formal behaviour is one that is explicit and precisely defined. These senses are related, in that presentation of an explicit definition of a code may lead to greater stability.

This research is concerned with formalizing in the sense of describing rigorously how actual notations are structured, thereby making it possible for the activity of notating to be carried out formally. It does not mean to prescribe that all pictorial communication *should* be strict and formal. A graphical notation may be intended for formal use, though it lacks any express

definition; conversely, a notation may be informal in style, yet we may still seek to describe it precisely. Precise description presumes that a code (whether stable or not) can be given a basis in some logical framework. Any mathematical model of natural phenomena must, however, be an idealized or simplified system – as a human artefact, a notation can never be precisely defined. The act of formal description thus inevitably prescribes some restrictions on any notational practices that rely on its support.

For simplicity, the idealization here will largely ignore questions and details of context. The usage of expressions involves method protocols, shared knowledge of the working context, and general knowledge relevant to the domain. To formalize this would require a model of the complex information processes of software development methods, as well as the cognition of the participants and the functions of the tools used to build software systems.

A further simplification is to ignore historical processes. Linguistics distinguishes two aspects in the study of a language (Saussure 1916): A *synchronic* approach looks at language structure as it exists at some point in time; a *diachronic* approach describes the processes of change and evolution which languages undergo. This thesis follows the linguistic tradition in treating notations synchronically, idealizing their instantaneous structure, and not attempting to understand how they develop over time, or how they are learned and used.

4.1.2.3 Informality and Ambiguity

The intended formalization will also disregard the possibility of *informality*. Notation is *informal* when its rules of interpretation are not fully agreed or understood – which may lead to errors of interpretation that cannot be automatically checked or converted. Informality is pejoratively referred to as a source of *ambiguity* and *vagueness*. Against this, we learn from Design Theory (Lawson 1996) that these attributes, typical of natural language and rough sketches, have an important function in the early stages of a design project, not only because they defer decisions, but because they help the imagination to play its part in searching for resolutions.

This suggests a need to support a refinement process that progressively removes ambiguity, allowing for transitions between informal and formal representations. Just as understanding of a software system must become more precise as design proceeds, informal expression must give way to formality at whatever level of abstraction may be required. Formal notations then provide

the bridges needed to connect the human domain of informal knowledge with the computational mechanisms of accurate reasoning.

It may be possible to give a formal treatment of ambiguity, perhaps as a set of coherent interpretations of an ambiguous expression. Ambiguity is not the same as non-determinacy, hiding of detail or successive approximation, which are acceptable and formalizable ways to defer decisions. The criterion is that a viewer be *unaware* that an alternative or less exact interpretation exists. Ambiguity has the potential to stimulate many alternative meanings – a kind of implicit logical disjunction.

These observations suggest that the imperative for formality may lie in the *prevention of ambiguity*. It follows that formal visual symbols must be clearly recognizable and associated by mental habit to concepts within a stable cognitive model. If this model can be simulated by abstract symbol manipulations, it might be possible to specify an unique meaning for every well-formed expression.

The existence of firm and explicit syntactic rules does not suffice to prevent ambiguity, which is a property of *interpretation* of syntactic form. On the other hand, avoidance of ambiguous expressions does not preclude representation of logical disjunctions, existential propositions, or even fuzzy predicates, all of which are matters of semantics.

4.1.3 Notational Roles

In order to keep in mind the uses of notations reported in the survey of Chapter 2, it will help to make a short analysis and classification of roles that they play. The discussion considers the ways that representational expressions serve in dealing with the structure of complex systems and other aspects of systems development. What do people do with notated expressions?

4.1.3.1 Instructive and Engaging Notation

Martin & McClure (1985 p109) describe diagrams as aids to clear thinking – if only one person is developing a system – and essential to communicating when several people collaborate.

> "A formal diagramming technique is needed to enable the developers to interchange ideas and to make their separate components fit together with precision."
>
> "A poor choice of diagramming technique can inhibit ... thinking"

A notation fills an *instructive* role when it is a tool for thinking: extending memory and aiding imagination. Such notation can be personal, invented as needed and informal at first, but the desire for rigour may lead to formality. The ability to support informal reasoning requires an accurate use of structural analogy, implying that pictorial expression can help. An instructive notation must be sufficiently formalizable in order to have interpretive rules that a user can learn.

The notion of *engagement*[1] is useful here: an engaged viewer is absorbed with content of expressions, not their superficial form. Expressions are *engaging* if they are easily read and understood by a diverse group of users. Engaging styles reduce the amount of cognitive investment needed to acquire competence with a notation's syntax and rules of interpretation. Engagement is supported by using features familiar in another context – a different notation or general perceptual tasks – because the effort required to learn a notation depends on previous experience with similar graphic, syntactic and semantic structure. Direct features of diagrams and words help to draw viewers into the world described in a notation, as a result of general pictorial and verbal cognitive skills. Easy engagement in using a notation relies on firm habits of interpretation that lie below normal levels of awareness.

The user becomes engaged in the notation only when the rules of interpretation are internalized; when learning a notation, the user is still partly absorbed by its syntax. Full interpretation is only possible when the user understands the subject domain for the notation. The degree of subject-understanding is an important criterion in choosing suitable notation. Experts in their subject can then engage with very complex notation; Novices in the domain may require a form of notation that is both *engaging* and *instructive* in helping to learn and understand the subject.

The expert user may wish to modify the syntax to make it more concise, or to better express new patterns in the subject domain. In exceptionally difficult work, the invention of new notation may be the first step in gaining insight into a problem, moving towards a more precise understanding that can be communicated rigorously.

4.1.3.2 Formal Notation

The need for rigour in representation favours compact notation over narrative text; concise notations can also serve as vehicles for computation. Where precision is important, a *formal*

---

[1] See Brenda Laurel (1991a,b) for a discussion of this term as an application of dramatic theory to human-computer interaction.

mode of expression is desired in order not be ambiguous. A notation has a formal role if its meaning is subordinate to its rules of manipulation. This possibility of reasoning by transforming symbols is opened up because expressions arre held in a persistent medium, giving aid to long-term and working memory. Expressions are in a code that may be processed by a machine or person as calculator, as in algebra and arithmetic. In computation, only formal operations are carried out on expressions.

Formulae can be compact and easier to use formally, but people need considerable skill to manipulate them. Strict rules of syntax and semantics must exist, but need not be explicitly known by a (human) user.

For example, programming languages are formal codes and tools for computation. Despite their formality, as Hoare (1986) observes, programmers are unaware of the laws that the codes obey. Though each language has its own community of users, they do not primarily use its code for general communication or instructing people. To overcome this, programming style encourages practices such as copious annotation in natural language text, and the use of familiar words as formal names. The predominance of individual use may be the reason for the wide variety of codes available. In mathematics, whenever formal expressions are also used as communications, there is much more uniformity.

Programming is a difficult task that is not much helped by textual codes that imitate a restricted form of natural language and which force preciseness of form before accuracy of content. All established codes are practically formal, with strict syntax and implicit semantics by virtue of compilation and execution by machine. This operational formality is not explicit (beyond syntax), and hence not easily available to programmers, although it can be explicitly mathematically defined, using denotational semantics, as advocated by Hoare (1986). There are rarely any software tools available for semantic checking and processing, though some provide animated execution.

A programming notation is instructive only insofar as it gives insight into the nature of computation; it needs to be easy to engage with if novices are to be involved. Attempts to devise codes that are more 'natural', either with linguistic theory or with graphics, risk losing the formal precision, and do not necessarily make them more expressively accurate. The redesign of

programming languages along "structured" and "object-oriented" lines, sometimes guided by mathematical analysis, has improved their ability to instruct and communicate: attributes that are also found in the corresponding systems-diagramming notations.

4.1.3.3 Roles and Activities

We can analyse roles according to activity, the agencies involved and knowledge needed for the activity or use of expressions. This is presented in a table[2]:-

Role:	Activity:	Agency:	Requirement:
instructive	**Thinking**	individual	own knowledge
engaging	**Communicating**	community	common experience
formal	**Computation**	calculators	simple, strict rules

We see that notation is used for quite different purposes on different occasions, implying varied requirements on style and structure. Before questions on specific problems of style can be posed, the mechanisms that make expressions meaningful must be examined.

4.2 Exploring Semiotic Theory

We have gained clarity on the concepts of graphical notation and formalization and identified three roles that technical notations variously fulfil, which are related to activities performed with software design representations. We would like to build notations as reliable bridges from informal thought to precise reasoning. Can a mathematical theory of symbol systems provide the technology for this?

In order to place technical notation within the wider background of sign systems, this short section outlines and comments upon basic notions and terminology of general semiotic theory as presented by Umberto Eco (1976). We follow Eco's division of the subject into a theory of *codes* which govern the behaviour of signs, and a theory of *sign production*, that concerns how a signs acquire meaning. The indented commentary points to notational examples of the concepts.

4.2.1 Signs and a Theory of Codes

Signs in general are governed by a theory of *codes* (Eco 1976 ch2), such as those that control the

[2]We should not assume that interaction with a computer counts as communication, nor that knowledge of context is necessarily coded.

structure and functions of notations. We consider the composition and purpose of general codes.

4.2.1.1 The Function and Content of Signs

A ***sign***[3] is defined by its ability to stand for something else: its *interpretation.* The relation between its form and its meaning may be contrived or natural, but must not be an identity. It is important to remember that signs are defined by this relationship, not by any intrinsic structure of the objects that embody the sign.

> In notation, our concern with structured items is only in regard to their role in communicative or interpretive acts. Eco's semiotic theory does not address use of sign systems in thought, but we take it that semiosis should include thinking, where expressions stand for *ideas.*
>
> For example, an interpretation of program code can be the *idea* of what it does: its specification or its potential for execution by computer. Yet this code is mostly not 'uttered' as a communication to other people.

4.2.1.2 Signs and Codes

A sign system consists of an ***expressive plane***, correlated by convention to various ***content planes***, including a *primary* content plane. The structure of a convention is known as a ***code***. Codes define these correlations by means of a system of ***sign-functions***, each of which establishes the correlation of a ***sign-vehicle*** (a *signifier*: an abstract element of the expressive plane) with a ***sign-content*** (an abstract element of the content plane: a unit of meaning).

A sign-content is known as an ***interpretant***[4]. It is an abstract entity: a *cultural unit*, not a real object. Thus a particular culture 'owns' the code and the units of meaning in content planes. These units may be analysed by their types and features, which are further elementary cultural units, called ***semic*** attributes.

The direct interpretation from expressive plane to primary content plane is determined by the *primary code*, which is concerned with ***denotation***. There may be a second code that correlates units in the primary plane with interpretations in a second content plane. This gives rise to an indirect interpretation of expressions in the second plane, known as ***connotation***.

We can infer that semiotic codes control kinds of tenuous logical connexion, which are temporarily

---

[3]Terms in bold italic are those used by Eco.

[4]It is sometimes called a *referent*, though here we prefer to reserve this term for concrete items denoted in context.

and approximately maintained by cultural (and sometimes natural) agencies. Where there is a *system* of notations and contexts, the connotative meanings can spread out into many domains.

> For example: the use of Ascii binary code sequences to denote a sequence of decimal digits which in turn denote a number, say; a number such as 2000 may denote a certain date, which may acquire the final 'connotation' of a millennial problem.

A code establishes sign-vehicles from which concrete ***tokens*** are generated. Eco points out that ***replicability*** of tokens is important. Signs are manifest as physical objects or events, which are in some cases difficult to replicate, or even unique, like cultural events or works of art. Formal Graphic expressions lie at the other end of the scale, since they can be indefinitely replicated by printing or writing. The copying of symbols is aided by ***articulation*** into combinational units, which is a feature of notation in mathematics, and computer graphics. These units (e.g. screen pixels) need not be individually meaningful.

> Expressions (or terms) in a notation are sign-vehicles, and their manifestations in print or electronic form are tokens.

4.2.1.3 Combinational Rules

A sign-function can be defined in its own internal structure, and in relation to its combinational possibilities within a context. Combinational rules are grammatical properties of the sign-vehicle, independent of its function. They portray expectations of connectivity between types of symbol in an expression.

Not all signs are *articulable* in this way. Spoken language has a characteristic "double articulation" of sentences, firstly into morphemes and secondly phonemes. The elements of the second articulation have no meaning in themselves; their forms have *oppositional* value, i.e. they have distinguishing perceptual features.[5] Morphemes sequence the phonemes into meaningful units.

> Similar structure is often found in textual notation. In other notations it can be unclear how to carry out such an analysis. Mathematical formulae are mostly singly articulated, with each character having meaning; diagrams may be resolved into textual forms, shapes, primitive graphic items (lines and circles), and perhaps pixels.

4.2.1.4 Semantics and Pragmatics

The semantic content plane is organized as a system of ***sememes*** (units of meaning), each

---

[5]Could these features be seen as constituting a further lower level of articulation?

occupying a distinct place within a semantic 'space'. The sememe presents all coded denotations and connotations as a function of context, and is thus not itself an item in a content-plane.

Whereas semantics treats abstract expressions, pragmatics deals with sign-tokens and signifying acts. The semiotic code also determines how expression usage depends on context, when this is analysed into its cultural units. This dependence on context is described by various types of presupposition: using an expression is said to ***presuppose*** certain properties of its context.

Eco describes several kinds of presupposition, all of which can be found in notation.

1) Referential	– presence of a reference for each name used.
2) Contextual	– logical compatibility with neighbouring expressions.
3) Circumstantial	– what participants need to know about context.
4) Semantic	– metaphors or other temporary meanings in operation.

In a software development context, (1) and (2) are required for checking acceptability of an expression being enacted. Circumstance (3) is more difficult to accommodate, since it requires keeping track of the discourse, and modelling participants' knowledge. Use of creative metaphor (4) is a feature of informal notations which might aid a discussion of requirements or design solutions, but cannot usefully be formalized.

Pragmatics essentially defines a relation between expression and context that determines which symbols may occur and that restricts future actions, reactions or responses. Expressions can be effectively enacted only if their context supports it. This presumes that some of the complex context has been analysed and encoded by means of a formal approach such as Situation Theory (§3.1.2). When an expression is enacted, presuppositional checks can then be carried out, and any ambiguity may then be resolved by reference to this contextual structure.

Contexts often take the form of a *discourse* between participants, or an extended exposition in a document. Discourse processes may allow new meanings to be attached to signs, symbols or parts of expressons.

The structural rules of *paragraphy* that apply to documents also have this feature (e.g. the 'definition before use' rule).

Such ***extracoding*** pertains to creation of a new code, and lies both within a theory of codes and of sign production.

4.2.2 Sign Production

Sign production (*ibid.* Ch.3) refers to processes which bring about new coding conventions, how signs and expressive structure come into being, grow and develop, how a code is established and

maintained; whether by natural or a formal process. The topic of sign production touches on diachronic issues which are relevant to the motivation and invention of new notations, or changes in style, syntax or semantics which may occur owing to change in skill of users or differing technical environments.

4.2.2.1 *Motivation* for Signs

The first concern is how signs and structure are motivated. To invent a content for a sign-vehicle, some motivation or stimulus must be present to adduce a correlation, which at length may be recognized as a new convention. The originator must elicit in the viewer perceptions 'equivalent' to those experienced in the actual idea. The three main methods of achieving this were identified by Peirce as:

> 1) arbitrary symbolic associations made familiar by repeated use,
>
> 2) associations motivated by metaphor or similarity, and
>
> 3) temporary reference made by an action of pointing to an item in context.

Peirce uses terms ***Icon*** and ***Index*** to describe signs motivated by (2) and (3) respectively. Eco regards Icons and Indexes as practical devices to create a sign where there is no previous convention. We can regard (1) as motivation by past *usage*, a general principle that applies to all signs.

> A notational example of (1) is Gödel's deliberate coding of logical formulae into natural numbers, constructed for his proof of the incompleteness of arithmetical theories. For (2), consider the Roman numeral IIII, resembling four tally marks, as opposed to the arbitrary numerals V, X etc. For (3), in elementary algebra, a variable 'x' which may "point" to a particular value in context; note that this relies on the existing arbitrary convention that x denotes the 'unknown quantity'.
>
> In notations, especially diagrams, the signifier may be a graphical relation between other tokens (e.g. insideness); hence meaning of structure must also be motivated. Iconic forms are common in more pictorial notations, but less common in formal ones.

Before meaning is motivated, pictorial combinations are *potential* signifiers: ***pseudosigns***, not signs; their structure may challenge a viewer to find meaning. We find this for instance in connotation of stories.

> Pseudosigns are informally present in documents describing the overall purpose of a specified software system. They are also a familiar challenge in mathematics, where it is customary to create formal systems without indicating any meaning for them – Girard in (Girard *et al.* 1989).

4.2.2.2 Indexes

The term 'index' means a sign such as the gesture of pointing. It can be seen as a form of temporary signification, where a referent is created 'on the spot' as the symbol is used. In notations an index is a name, tag or arrow that 'points at' or cross-references an element of the same or another diagram or elsewhere in the context. This ***deixis*** (pointing) is rarely literal, and often dependent on further convention or metaphor. e.g. letters on a geometry diagram, variable-names.

> Indexical methods include the use of an arrow to link two separate expressions, though in this case the arrow is also an icon for some notion of linkage.
>
> In formal language, **definition** is an important means of attaching meaning to an index. This must use an existing convention for defining, possibly in a different notation to that in which the sign will occur.

4.2.2.3 Icons and Structure

An ***Icon***[6] is a pictorial device that has metaphorical similarity to an intended semantic content. A notational example is a graphic arrow used to depict flow. Iconic signs are motivated by some *analogy*: i.e. an existing mental procedure that enables transformation from form to content. Strictly, pure iconism should be independent of cultural association; perhaps exploiting natural visual experience.

> In practice, the similarity must be conventional: Euler's circles rely only on a convention or metaphor which (arbitrarily) draws the reader's attention to similarities between spatial containment and properties generally. Even in Euclidean geometry, diagrams are not pure icons, because a convention establishes which graphical properties are salient. In these two cases, analogy is evident in the arrangement of symbols, whether or not they are icons.

Graphic arrows, and spatial succession, are examples of what Eco calls ***Vectorization***, which we may treat as a structural kind of iconism: a feature that contributes to the composition of an expression.

> We note that 'natural' perception is not the only source of analogies. Cultural experience provides a rich source of prior mental habits that can provide motivating analogies. These may even be semiotic habits themselves; a new notation may *borrow* a symbol from another familiar notation and use it in an analogous sense, e.g. + as a binary commutative associative operator; whole words are often borrowed from natural language. A related phenomenon in programming languages is 'overloading' of a symbol to generalise its meaning.

---

[6]This usage should not be confused with the 'user interface' sense of an ideogram or pictorial motif.

4.2.2.4 Strength of Coding

Iconic signs are often weakly encoded, in the sense that there is little established structure or consistency of usage. In a representational drawing, for instance, the signal is continuous, without recognizable articulation; it cannot be reliably analysed into signs or *figurae* with positional and oppositional value. Eco notes that its verbal equivalent is not a word but a whole story. Such ***weak*** codes are based on established example texts. In contrast, ***strong*** codes are based on grammar: known rules of combination.

> The notations we wish to consider are strongly coded: standardized and formalized to some degree.

4.2.2.5 Expressive Principles in Notations

To complete the commentary we can elucidate the mechanisms that produce coding in notations, in the light of this theory. What processes enable notated forms to be expressive?

This enabling function can be divided into three parts: signification must be motivated, defined and maintained. Motivation for signs is important in making notations easy to learn and use. Formal definition and the use of standard tools have should help to stabilize and strengthen the coding. Maintenance of signs happens by mental association and habit – regardless of how their invention was motivated. The success of notations then depends on how well their coding was originally motivated and defined, as much as how it is maintained. The ideas of Peirce and Eco suggest that the expressive principles can be summarized in three processes:-

> **Iconic Process:** invention by metaphor; borrowing sign or structure from another familiar notation or context, to exploit perceptual-graphical properties or even existing pictorial and linguistic conventions.
>
> **Indexical Process:** the act of pointing a sign to other entities in its neighbourhood, establishing temporary meaning. The connexion between the index and its referent is newly made, but the fact that a sign is used as an index (a holder of temporary meaning), must itself be already established.
>
> **Symbolic Process:** the repeated usage of a sign, establishing permanent meaning through association.

An index is a pointing function *temporarily* bound to context, whether this role is motivated by iconism or not. The definition given here regards a symbolic sign almost as a permanent kind of

index, which gains meaning by being repeatedly pointed at the same referent.[7] This is compatible with a definition of 'sign' as an object that *metaphorically* points to its interpretant, i.e. its abstract referent.

Each of the three processes has a particular task. Iconism is vitally important in signification; the argument leads to the position that iconism in its various forms is the *only* design feature that need be considered as a motivating principle. Secondly, in an explicitly formalized notation, an *indexical* process is used to *define* the meaning of signs. Thirdly, a *symbolic* process is a maintaining principle when the same potential vehicle is, in usage, always associated with the same interpretant.

We have now arrived at a set of concepts and processes which can guide us in mapping the inner workings of various styles of expression.

4.3 Aspects of Notation

We are now in a position to use the semiotic framework as a means of clarifying the folklore that surrounds notations. This section thus investigates notative methods and discusses the structure they exhibit in terms of the concepts just introduced. It first concentrates on diagramming as the main mode of organization. Textual form is then shown to be a subsidiary aspect. The structure of notation is summarized as consisting of layers of syntactic patterning that are employed to encode meaning. In this way, the discussion establishes the determinants of structure in the notations that we wish to describe formally.

4.3.1 Semiotic Characteristics of Diagrams

Here we look at coding methods found in diagrams, or rather in expressions which are not constrained to sequential form. Diagramming comprises a set of techniques used to produce signification in graphical expressions. We address the following questions:

---

[7] For example, the character /c/ may be used as a numeric variable representing the value of a speed that is indicated in context. If we repeatedly use it to denote the speed of light, it loses its function as a variable, and takes on a fixed meaning.

What is the expressive plane?

How is the sign-vehicle constituted?

What are the methods of coding?

How is the code articulated?

What are the content planes?

What are the units and attributes of meaning?

4.3.1.1 Elements of Code Structure

First we look at the kinds of graphical properties that are employed in diagramming. The "expressive plane" of graphics is simply the arrangements of marks drawn on a geometric plane. How is coding achieved in such two dimensional sign-vehicles?

Expressions are commonly viewed as being composed of drawn units of various sizes belonging to a hierarchy of constructed syntactic types. Expressions can be explained as arrangements of units, subject to attributive and locative relations. A whole expression is a unit of a certain type (e.g. 'sentence'). Expressions need not be homogeneous in style; they can be hybrids of different styles of sub-notation, each of which serves as a syntactic type. Thus diagrammatic notations mostly include textual components.

The units are drawn and *located* in an expression. More generally, items that form expressions fall into three groups:

(1) lexical units such as keywords, box shapes, links;

(2) attributes of size, orientation, colour, texture; shape, symmetry, markings on single items;

(3) instances of spatial relations such as *near*, *above*, *touching*, *inside*.

Lexical items belong to a finite set of types – e.g. characters, character sequences (names and numerals) or pictograms. They are often used as what may be called *pictive* signifiers – items which use geometric shapes that are perceived as a pattern, requiring skills of recognition and discrimination. They need not be pictorial in the sense of directly resembling some other entity.

Attributive relations may be used to create a hierarchy of subtypes, embodied as variants of lexical items. They may also appear as semantic modifiers, for example using crossing out to express negation, deletion or prohibition. Attributes give rise to a finite set of graphical features that units may possess.

Locative relations are a major feature of diagrams; they usually involve some *structural* analogy.

The topological notions of nearness, continuity and connectedness help code the combinatoric properties of this structure. Spatial relations between shapes were noted as important for defining graphical syntax in the previous chapter (§3.1). Although such relations are frequently referred to as topological, this is not the full story; geometric cases such as collinearity and horizontal / vertical orientation are also needed.

A small number of 'qualitative' spatial properties appear to cover most kinds of diagram incidence. It is interesting that the Euclidean plane geometry of the page, the properties of length and angle, are little exploited analogically in the notations we have chosen to consider. Syntactic units may make extensive use of recursive constructions that could in principle generate expressions of arbitrary size and complexity. The coding restricts such recursive freedom so that cognitive limitations can be respected. We assume that the coding is discrete, so that we only need to consider finite expressions. The use of recursion is essential to a certain notion of syntactic *abstraction* that will be discussed shortly.

This analysis should not, however, be taken as complete. There seems no certain limit on the types of pattern or relation that are used – creativity in notation may lead to others not yet invented.

4.3.1.2 Diagramming Techniques: the Diagram Body

What are the combinational rules found in coding of diagrammed notations? The various structural mechanisms employed in software development diagrams are described by Martin & McClure (1985) and Nickerson (1995). The latter offers a 'thorough survey' of the use of diagrams in computer science and identifies some underlying principles: visual conventions such as Adjoinment, Linking and Enclosure. Although we do not have the benefit of an exhaustive study of the many examples of diagrams, we can at least explore such mechanisms in more detail, from a general viewpoint.

Here an analogy from the biological world may be helpful, in which we view a diagram as having a *body*. Diagrams often employ *enclosure* and *linkage* – forms of box and arrow – that function as a skeleton or *frame* to which labels in another notation are attached. These often serve as kinds of 'shells and bones' enclosing and connecting portions of text: enclosures form an 'exoskeleton', while links form an 'endoskeleton'. These skeletons have *connective* or *collective* function; they

may either accept collections of items, or they may have fixed attachment areas which must (or may) each be filled by a single item. Some examples of enclosures are text bracket pairs and the schema boxes of *Z*; enclosures that allow overlap are found in Venn diagrams and Higraphs of (Harel 1987, 1988).

Framing defines an association of places (*slots*) to be filled by symbols, like the position before or after a character, or a space between separators in text. In formulae, super- and subscript positions are slots (and filling expressions are often reduced in size); numerator and denominator are slots in fractions. A slot may itself be filled by frames, leading to a hierarchical nested structure, as found in Entity-Relation diagrams and Nassi-Shneidermann charts. Slots may be organized geometrically: tables and matrices have their slots structured into rows and columns, exploiting the two dimensions.

4.3.1.3 Expressive Mechanisms

To summarize: Graphical construction employs several mechanisms including:-

Pictive	– sets of alike items, labelling
Attributive	– modifications, markings or colourings
Locative	– adjacence, apposition and sequence
Collective	– framing, enclosure
Connective	– polyvalent linking
Quantitative	– repetition, size and shape

In notation design, the choice between these is related both to subject domain and practical constraints on size and complexity within the representing medium. The mechanisms are used to support many analogies, including the forms we have easily referred to as "structure": the *building* of expressions. Coding may also be motivated by metaphors with other domains of experience, cued by spatial or textural properties of the embedding in the 2D physical medium.

For a wider view of graphical representation techniques, the reader might see the range of examples in (Bertin 1982, 1983; Tufte 1990) – illustrating the full variety of visual information formats that people have devised.

4.3.1.4 Semantic Structure

The content plane for a notation may be some kind of design representation concerned with software systems. In designing a notation it would be necessary to consider the semantic domain

in detail. Here we only look at the motivation for diagram semiosis in general terms. How does diagram structure carry meaning? Analogy and abstraction are the key techniques here.

4.3.1.5 Specificity and Quantity

The iconic quality of diagrams has been noted previously (§2.1.3). A good example of this is the way diagrams may represent quantity, in the senses of multiplicity or size, by a proportional mapping of lexical items to individuals. In this way, Histograms, Piecharts (and coordinate-geometry graphs) are used for coding numerical values, when the size of a line or region is proportional to the size of a set or space. In the discrete forms of diagram considered here, the one-to-one representation of quantity is sometimes useful. In contrast with decimal numerals, this analogical coding can be seen as a weak instance of *specificity*, a phenomenon that was described in (§2.1.3).

The notion of specificity is explained in (Lee *et al.* 1991):

> "[graphical representations] are limited to representing total mappings of identity and some spatial relations... This totality of mapping is what allows efficient access by search mechanisms. ... The specificity hypothesis is that this property of totality, shared by images and the relevant components of working memory, is what gives graphical representations their special cognitive properties."

Stenning & Tobin (1994) thus regard the property of specificity as the factor that yields tractability of inference in diagrams. They use specificity as a criterion to discriminate between diagrammatic presentation and languages, noting that most graphical systems employ more than one directly interpreted relation – though specificity is also found in "degenerate" textual languages where concatenation is directly interpreted as temporal succession, for example. An inherent disadvantage is that the diagram cannot hide the relations between individuals that are signified by spatial arrangement – a property that Stenning & Tobin call '*information enforcement*'.

We can view specificity in diagrams as a principle of *structural iconism*, which motivates aspects of the code structure and so provides a stronger motivating principle than the simpler iconism of pictorial symbols. The main example of this is the use of one-for-one mapping between a set of similar lexical items and a represented collection of individuals. The diagram indicates not just the number of individuals, but usually also the connections between them. The cognitive advantage of specificity is lost when a set has too many members (e.g. graphs with many crossing edges, or spread over several pages).

4.3.1.6 Modularity and Abstraction

According to (Lee *et al.* 1991), languages characteristically allow abstraction in their coding, in contrast to the specificity found in diagrams. This abstractive power occurs at the expense of processibility. Despite this, we find that abstraction is also an important feature of diagrams in software development.

The reason for loss of directness in notations lies in the complexity of software systems. When a diagram becomes complex, the viewer investigates it in the manner one might explore the workings of a machine – some notations (e.g. electrical wiring diagrams) are not intended to break down into sentence-sized pieces. To avoid this kind of intractable layout, it is common to extend the notation syntax by using modularity (as in program text) or abstraction mechanisms.

Modularity provides a way of "covering" a large system by separate small expressions that are linked by common references. As a strategy for communicating complexity, it draws the attention of the viewer to facts and factors of the system in small 'packets', such as modules, sentences, statements, or diagrams. A collection of such packets describes a possible system only if it is consistent and coherent according to semantic rules. Each member of the collection may be implicitly linked to others by sharing of name-items that have a common referent. Simplicity is maintained because in any particular notation, only certain aspects of a system are revealed.

Abstraction can be seen as a kind of modular technique, that involves replacing a complex subunit by a single name-symbol, whose definition appears as a separate expression. This creates a hierarchy of dependent definitions. In recursive definitions, the hierarchy is not well-founded – it contains cycles. This kind of recursive abstraction allows a finite expression to denote an infinite structure. The cost of this abstractness is clearly that some computation must be carried out in order to decode the expression.

The cross-referencing needed to connect modules requires a many-to-one mapping from such abstract lexical items to their shared referent: a name occurring several times refers to the same individual. This 'type-referential' behaviour provides an alternative to links, and is therefore important in textual notations, where connecting lines cannot be used.

4.3.1.7 Graphs of Many Types

Graphs are structures that lend themselves to both abstract and direct forms of coding. In Chapter

3 it was found that diagrams are often regarded as drawings of graphs of some kind. This is implied by McWhirter (thesis: 1995), who observes that all graphs are *abstract* structures, whilst graphical notations of any kind are *drawings* which embody them.

If the domain being represented has already been formalized as a class of graphs, a notation may be chosen to realize these graphs by pictorial shapes and features, which make up the skeletal structures referred to above. The incidence between parts may be realized in different ways. There is however a close correspondence between the constraints defining a graphs of the domain and the modes of graphical realization that are possible. Thus a structure of nested enclosures realizes an acyclic 'tree' digraph, and a planar graph may be drawn without crossing edges.

These kinds of graph (referred to here as *graphtypes*) are distinguished by their permitted methods of connection. Thus edges may link ordered pairs of nodes (as in directed graphs) or tuples of nodes (hypergraphs) or unordered sets (as in webs); the number of edges connected to a common node may be limited or fixed; edges from a single node may be sequenced – for example, flowcharts may be described as hypergraphs in which boxes are edges between junctions of arrows. There may be different kinds of nodes (e.g. places and transitions of Petri nets), or of edges. Edges may link other edges, in a more general *relational structure* that goes beyond the usual graph paradigm, as found in Entity-Relation Diagrams. As well as these simple forms of incidence restriction, more complex logical constraints on graphtypes may be applied, such as connectedness, absence of cycles and planarity.

4.3.2 Textual Form and Structure

The familiar example of graphic text can help illustrate the concepts just discussed. This will be rather a matter of stating the obvious, in order to show how many 'diagrammatic' characteristics are already present in text. For concreteness, we consider the case of a textual message laid out in a rectangular region, and ask how the structure of such a 'message notation' is constituted.

Pure text has an easily described uniform visual structure which serves as a carrier for much more elaborate and varied syntactic patterns. It is structured as *strings* of *elements* of a set, called its *alphabet*, consisting of graphical shapes of similar size. These elements may themselves be discriminated by visible features of shape or structure (e.g. diacritics, underlining) – but in this case we exclude non-sequential mechanisms such as subscripting.

4.3.2.1 Graphical Structure

A message is drawn as spatial sequences of shapes (called *characters*) in a particular orientation, spatially arranged in sequences parallel to the base of the rectangle. It is assumed that vertical 2D layout of characters is not significant, since a simple textual notation is intended. It is then possible to abstract this geometric aspect of the graphical layout, and concentrate on the nature of a message as a *character string*.[8]

Characters are *stylized*; each is displayed by one of a range of perceptually equivalent close variants. In detail, elements have graphical forms *characterised* by features that allow perceptual discrimination between them. Thus ambiguity may occur if two distinct characters share all the same features. This could happen in three ways:-

1) shape: /O/ confused with /0/, (similar features)
2) visible structure: /rn/ confused with /m/. (between an element and a compound)
3) overloading: unary negative /-/ confused with binary minus /-/ (identical characters).

These kinds of geometric-perceptual ambiguity may be resolvable by syntactic rules, making use of the context of the characters within an expression.

4.3.2.2 Articulation

The alphabet is a small finite set, not a recursively generated set of shapes. It forms the lowest layer of articulation: the smallest kind of unit. The alphabet usually contains different kinds of 'pictive' elements such as letters and punctuators.

The second layer of articulation is formed by composing character shapes into lexical units, e.g. letters into words. This concatenation is a simple recursive process, but only a finite set of lexical signs (a vocabulary) may be admitted in expressions. Perceptually, the lexical units are recognized as wholes rather than combinations of parts. Words may have graphical attributes, such as lower / upper case, boldness, choice of font.

Letters are character shapes, mostly with no individual meaning. Letters are concatenated to form *words*, generally the smallest items that can be given individual meanings. Some single characters also serve as lexical units, and may also be distinguished by significant attributes, e.g. the negation mark on the inequality sign ≠. Some strings (names) may be extensions to the

---

[8]Characters include notional spaces, which are not elements of the alphabet.

vocabulary – newly defined by an indexical process. Some strings (e.g. numerals) have an internal syntax and semantics determined by a sub-notation.

Punctuators are syntagmatic items for linking and enclosing, e.g. the separators and brackets, which are involved in both articulatory layers, and form the 'skeleton' of the text. A bracket pair must then be regarded as a lexical unit, though not a word.[9]

If the messages consist of standard phrases, this would constitute a third articulatory layer. Special phrases that have more meaning than their internal syntax suggests, are examples of *overcoding* in Eco's terminology.

4.3.2.3 Syntax

Above these articulatory layers, the syntax (typically) becomes recursive in its complexity. Whereas the graphical and lexical layers are bounded in size, the syntagmatic layer creates (indefinitely) larger, complex units whose meaning is derived from its components, e.g. terms, phrases, clauses, sentences, and also *paragraphic* units such as blocks of statements. Here text differs from diagrams in having little skeletal structure to delimit the units. For example, in algebraic formulae there is a tendency to rely on implicit groupings where possible – such as infix operators. This can cause ambiguity. On the other hand, over-use of punctuators leads to awkwardness (e.g. the parentheses in LISP code).

Syntactic conventions of text are broadly unaffected by semantic structure; their purpose is to overcome those restrictions of the medium that conflict with a more direct representation. In natural language text, syntax is mostly concerned with explaining syntactic groupings as arising from implicit lexical features, as is evident in categorial grammars and feature-based approaches (see §3.1.2 and §4.4.2).

4.3.2.4 Semantics

The top layer of structure is the most abstract form of syntax – a conceptual semantic structure that is effectively independent of graphical format; e.g. an algebraic formula can be represented by a directed graph. The semantic layer is a formalization of the structure of denoted objects in

---

[9]Where brackets cannot be paired perceptually, they fail to constitute an effective lexical unit, and comprehension may break down.

the content domain of the notation.

These upper two layers of structure cannot qualify as kinds of articulation, because the units that compose an expression are overlapped in a complex manner. A recursive syntax can admit an indefinite number of units, covering an unlimited range of meaning, though they belong to a finite set of types.

4.3.2.5 Motivated Signs

Various limited kinds of pictorial iconism can be found in text. A few character shapes, especially in formulaic text, have visual iconic features based on geometric shape or symmetry, e.g. /=/ and /</, or the brackets ( ) [ ], whose syntactic function is suggested by their shape. Some symbols such as digits and /+/ are borrowed, typically from arithmetic – the borrowed icon /=/ has an almost universal status. In programming texts, keywords are borrowed from natural languages, and fragments of natural interpretation are converted by metaphor to the more formal meanings that a programmer understands – as if a formal denotation corresponds to an informal connotation. Structural iconism (specificity) is present when the sequence or adjacence of substrings denotes (temporal or spatial) ordering of referents.[10]

We see from this brief description that text has available all the features found in diagrams, though within the restrictions of its sequential form.

4.3.3 Modalities, Mechanisms and Layers

We end this investigation of semiotic aspects of notation by considering the differences between modalities or styles of expression. These are characterized by reference to the mechanisms of pictorial, verbal, linguistic and spatial coding. To round off, a description is given of the various semiotic layers found in notation structure.

4.3.3.1 Diagramming versus Text

In what way do diagrammatic mechanisms differ from textual?

Modality and *style* refer to characteristic types of coding and convention of use. Diagrams, text

---

[10]This iconism of succession is rather weakened by the graphical breaking of text into lines, because it then relies on cultural conventions of reading to determine sequence and direction.

and formulae are distinct in style. We look for some clearer criteria of classification for notation styles than those used in the Shu Triangle (§2.1.2) for visual language styles.

The above inspection of text reveals four or more structural or general *syntactic* layers: graphical, lexical, syntagmatic arrangement, and at least one semantic layer. As remarked above, diagrams show a less strict separation of the layers. For example:

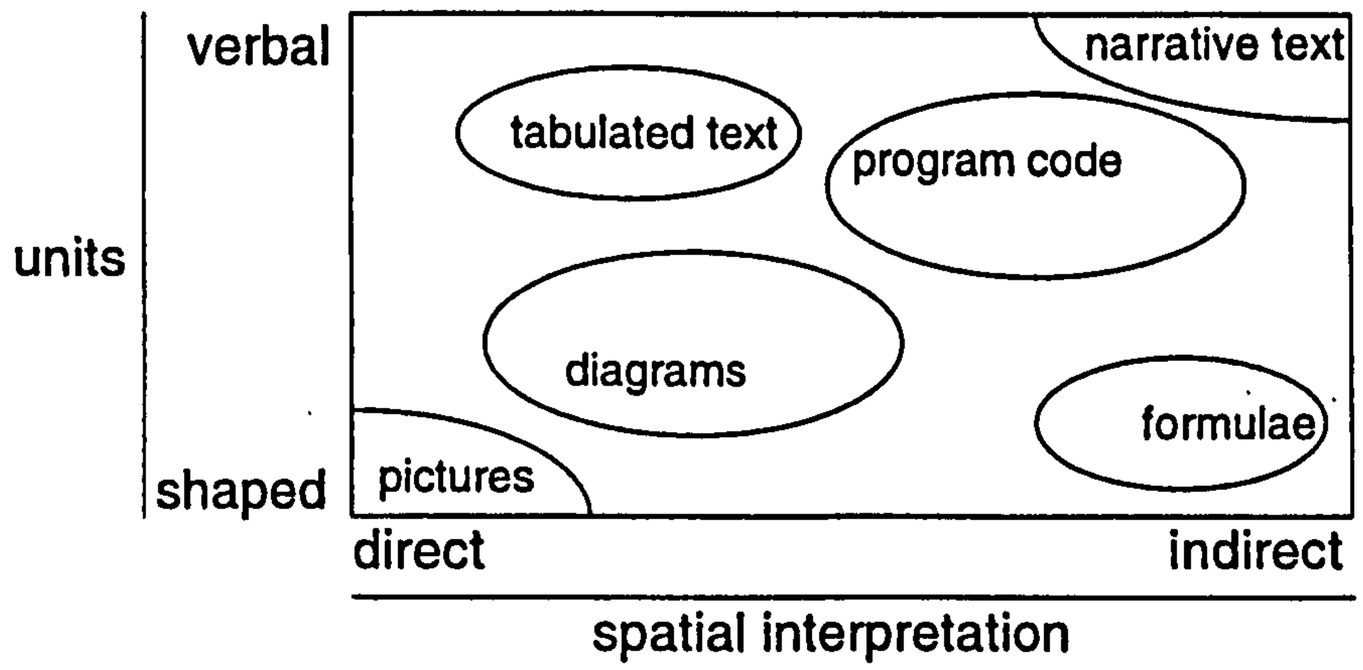
- Diagrams have a greater reliance on 'collective' lexical items that explicitly frame larger units.
- Operations such as type checking are treated as semantic in textual programming languages, but are part of connective syntax in visual languages.
- Diagrams use structural analogy and visual metaphor-cues to make the semantic concepts correspond directly to graphical processes.

Text has a clearer articulation into layers owing to the linguistic processes that have evolved to overcome the restraints of sequence.

There is also the the case of **formulae**. Formulaic (e.g. mathematical) styles are intermediate between text and diagrams in these respects. Comparing formulae with text, we find less reliance on sequence and more use of the vertical dimension, with larger graphical separators and frames (e.g. matrix brackets, fraction lines). The lexicon may be almost absent, since special (pictive) characters are often preferred to words. Formulae make no use of the long graphical linkages found in diagrams.

4.3.3.2 A Method of Classification

This suggests an approximate way to classify notational styles – according to the extent to which certain expressive mechanisms are used in the layers of semiotic construction. The method is based on two questions: Are the smallest significant units pictorial or verbal? Is the spatial arrangement of these units directly significant or indirect – abstractly interpreted by some recursive linguistic process? These questions suggest two average ratios, of verbalness and directness (specificity) in expressions, which can be used as dimensions. The chart below accordingly places various styles of notation in a rectangle.



From the chart we note that notations tend to lie on the diagonal joining the two 'natural forms' of narrative and pictures. It is less usual to find a spatial layout of isolated words; mathematical formulae are composed of shaped characters which are mostly not iconic. The use of words is a strong cue to expectations of linguistic syntax, whereas special characters in formulae give no cues to their structure for the non-mathematician.

4.3.3.3 Layers of Syntactic Structure

We now have an answer to the questions of (§4.3.1), on how expressive and content planes are connected.

The coding of an expression is regarded here as a general syntactic stratum that may helpfully be divided into layers. These layers of articulation, recursion, denotation and connotation are seen to mediate between the 'ground' (physical reality) of expression tokens and the 'sky' (meaning) – the essence of their presumed effects in the world. Each layer has its own logical characteristics, which are related to cognitive processes and resource limitations.

The two lower layers are articulatory. The graphical layer is a matter of simple perception – resolving a drawing into the spatial distribution of a finite set of identifiable shapes. The lexical layer consists of identifiable groupings of these; to be effective as communication, lexical items should be instantly familiar and recognizable.

The next layer consists of overlapping and sometimes extensive arrangements of lexical units. Parsing this central syntagmatic layer requires a search for structural matches, but may not practically take more than a second. The recursive procedures which underlie syntactic processing must be within feasible bounds for expressions of reasonable size – which is why grammarians are not content with grammar formalisms that are Turing-complete (Carpenter 1995b).

Fewer computational limitations apply to upper semantic layers. The upper boundary for the syntactic stratum takes the form of a semantic conceptual structure that can feasibly be deduced from the layers below. Beyond this, semantics may involve the unbounded complexity that unfortunately can arise in the relationship between thought and action. The full meaning of an expression may require complex logical inference; it may yield progressively more information over a long period. The full semantics of programming code, for instance, is a much more complex affair than its syntax, or even than its formal denotation. It concerns an understanding of what the program *does*.[11]

Semantic layering can reduce the logical burden; it occurs when metaphor is used to transfer familiar (perhaps literal) reasoning based on cues, into a more complex, less rehearsed arena of the actual subject domain.

In this way, semiotic principles determine how the structure of notation syntax is built up in layers and connected by expressive mechanisms.

4.4 The Problem of Defining Structure

The previous discussion has led us to the impression that notations possess a kind of layered structure. We now wish to concentrate on the question of how this syntactic structure can most appropriately be described. To find an answer, this section examines various kinds of explicit syntactic rules employed in linguistics. These grammatical and logical methods are then compared with semantic treatments of graphical analogy. Analysis of the methods points to the need to consider the underlying question of how to define the basic form of expressions. This question is resolved by an argument based on the formal operations that will be performed on the description.

4.4.1 Methods of Syntactic Definition

We begin by taking a look at the structural description of syntax. In view of the discussion in the previous section (§4.3), the term 'syntax' is taken to include in the whole of the layered structure of coding – not just the matter of arrangement of lexical items. How might syntax best be defined?

---

[11] In this sense it may be undecidable. Also, the exact nature and behaviour of computer systems has demanded a deep logical analysis in computing science and cannot be said to be fully understood, especially if we include parallelism in hardware and human factors.

4.4.1.1 The Nature of Syntactic Rules

A syntactic description for a notation specifies an expression as a well-formed form (wff). The *language* (expression-class) determined by the syntactic rules is simply the class of wffs.

Syntax can be taken to represent the viewer's mental model[12] of an expression that is divorced from any reference to its situation – as when taken out of context. A person's knowledge of syntax enables them to create or compose expressions. When an expression is received it also provides a starting point for the process of interpreting in order to respond.

Two broad approaches to structural description of syntax were noted in the last chapter (§3.2), with two corresponding kinds of syntactic rules:- .

1) ***Logical*** systems constrain a general structure to satisfy certain relations and properties.
– *Constraint* rules provide axiomatic criteria for accepting or rejecting a (generated) form. They are declarative rules by which forms are judged.

2) ***Production*** (rewriting) systems derive well-formed structures by a combinatoric method of freely applying local replacement operators to configurations of symbols;

– *Generative* rules build forms. They are executable rules which can convert a sequence of choices into an expression. During construction, forms are partial: incomplete or incoherent.

In a logical system, expressions are treated as **constrained forms**; rules do not specify how forms are generated, but only the constraining properties that determine well-formedness. When creating an expression, it is necessary to find a form that satisfies all constraints and also fulfils semantic goals – a problem which may be undecidable. When interpreting a form, the checking of syntactic constraints should be a quick 'decision procedure', and the process of discovering meaning is deductive. Therefore the syntactic rules direct our attention to logical form and to corresponding deductive and constructive processes. What exactly is meant by a *form*? For the moment it is assumed that forms are combinatoric objects belonging to some class called a *form-space*. The class of forms must contain the expression-class of a notation, since every expression is a form. We return to this question in (§4.4.4).

The form of an expression is not just a definition of its graphical appearance. It carries information on all syntactic layers – all aspects of form that are perceived, imagined and thought. Hence we

---

[12]This is not to claim that the viewer actually 'has' the model, but that they behave as if they might.

may speak of graphical form and semantic form. Form-spaces are necessarily different for each notation. The graphical form and semantic form of an expression are connected by a signifying relation.

In a production system, rewrite-rules generate wffs. The set of rewritings resulting in a wff builds a **derivation** for it; hence each wff has an associated non-empty class of possible derivations. When creating an expression, the writer is concerned with achieving a graphical form that matches an imagined semantics. In a grammar, rewrite-rules provide a way of navigating through form-space, which for restricted grammars will guarantee a well-formed terminus. Conversely, when interpreting a given graphical form, a search for derivations occurs; it may be undecidable whether the form is a well-formed – the search may fail. When the search succeeds, the constructed derivation is a representation of the meaning, from which it is possible to calculate a semantic form.

4.4.1.2 Heterogeneous Notations

We observed in (§3.3.1) that neither logical specification nor graph grammars have succeeded in providing a general and straightforward means of defining notation syntax. Why is this not as easy as the task of specifying formal textual syntax? This is because, as has been noted (§3.4.1), the graphical format of text is homogeneous: it remains much the same whatever the topology or topography of the domain being written about. In contrast, graphical notations have varied topography, which allows them to exploit spatial metaphors. How can we allow for the heterogeneous nature of notations?

We could enforce a common graphical form – a standard underlying structure for the whole gamut of existing notations, acting as an invisible stratum that can accommodate all possible metaphors. This would allow grammars to use production systems that are uniform in all graphical contexts. Yet even if we could formulate a sufficiently general graphical theory, it would introduce unnecessary complexity into the descriptions. For this reason, graphical mechanisms in notation cannot be formalized adequately by a single kind of production system. We cannot get round this by taking different rule-methods for each syntactic layer, because the distinctions between articulatory levels in notation are not very clear.

As was pointed out in (§3.1.1), the simplicity of textual syntax is illusory. If we admit formal syntax

that approximates the subtlety of natural language, structural description problems similar to those of graphics must be faced. Although text is graphically homogeneous, the semantic forms it contains can be as varied as for diagrammed notations. For this reason, there may be something to learn from linguistic approaches to syntax, which will therefore be explored below in greater depth.

4.4.1.3 Formal Grammars

If we look at simple formal grammars, we find that they do a more important job than just defining the class of expressions in a notation.

> A grammar is presented as a set of rewriting rules on an extended language of *partial* (unfinished) forms. These "P-forms" contain extra symbols ('non-terminals') that stand for syntactic types within the notation. In context-free grammars an expression can be analysed into a hierarchy of phrases, each with an implicit type. Generation begins with a single non-terminal, treated as a P-form. It proceeds by rewriting subforms as allowed by some rule and ends when no further rule applies. The resulting form is then complete and well-formed. Syntagmatic structure is represented by a set of equivalent derivation 'trees' that track the replacement of subforms. This allows the possibility of syntactic ambiguity; a given expression may admit two possible derivations that are not classed as equivalent.

The status of this process is somewhat mysterious, because it does not represent the human process of expression creation. Nor does a hidden derivation tree qualify as a direct semantic representation. This is not to say that notation grammars are unnecessary, but that their role needs clarifying. The main value in grammar is rather to provide operations that construct a consistent abstract syntagmatic layer 'compatible with' a given lexical structure.

4.4.1.4 Example: Binary Numerals

It is easy to illustrate that grammars are not primarily about defining the right set of expressions, but about deducing a higher level of structure from lexical form. This can be seen by examining two phrase structure grammars that *generate* the full 'language' of strings on a finite alphabet A = {0, 1} – the set called {0, 1}*. In this case the lexicon is the alphabet. Thus all strings on A will be well-formed, and will be interpreted as natural numbers.

> Let *S* be a start-symbol for two grammars G1 and G2
>
> G1 has three rules for constructing strings by appending characters :-

app0:	[S → S 0]	append zero
app1:	[S → S 1]	append one
nil:	[S →]	delete 'S'

The string /1010/ for example is derived as `(app0 (app1 (app0 (app1 nil)))` – here encoded as a LISP s-expression.

G2 includes a rule for connecting two strings:-

`put0:`	`[S → 0]`	place zero
`put1:`	`[S → 1]`	place one
`cat:`	`[S → S S]`	catenate 'S'
`nil:`	`[S → ]`	delete 'S'

/1010/ is derived as `(cat (cat put1 put0) (cat put1 put0))`, or in many other ways.

These two grammars define the same language, but with quite different articulations. G1 admits only one derivation term for any of its strings, which are reminiscent of stack data-structures. G2 admits ambiguity in strings as terms. If we treat all derivations of the same string as equivalent, G2 characterises A* algebraically as a *free monoid*, the associative algebra of sequences generated from *A* by the operator that concatenates two strings. There are clearly many other possible context-free grammars, each of which imposes some kind of term-structure on sentences of A*, if we read the rules as constructor functions.

The advantage of G1 is that it allows us to calculate the semantic value of the derivation, using the following arithmetic rules:-

$val[app0\ s] = 2 \times val[s]$ where s:S is a string.

$val[app1\ s] = 2 \times val[s] + 1$

$val[nil] = 0$

In contrast, G2 does not allow such a calculation because concatenating two strings does not correspond to a simple arithmetic process. Thus G1 is useful as a grammar for binary numerals, but G2 is not – even though they generate the same 'language' or class of strings.

It is instructive to examine the limitations of these grammars. The grammar G1 does not give the full interpretation into semantic form – the example does not define what the semantic representation of a number should be like. Neither does G1 specify the graphical form, as can be seen from the way these rules are notated. For example, the rule (`app0`) echoes the sequential structure of text. In the right-hand-side "`S 0`", the rule sequences the names '`S`' and '`0`' to indicate the sequence in a generated string. The letter '`S`' names the abstract syntactic type 'string', but the character '`0`' serves to name its own shape. Hence the rule-notation borrows the graphical form of the notation it describes. This is a circularity of definition, of the same kind as was noted in (§3.2).

Nevertheless it feels quite natural, since the rule is also written in a textual notation.

If we wish also to specify the sequential form, rather than take it for granted, we need to acknowledge relations of *precedence* which hold between places in a string – though not always in a literal graphical sense. This leads towards a *relational* grammar (§3.2.4), in which the characters in the alphabet become relations that link the *nodes* that correspond roughly to positions on the graphic form. The rules of G1 might then become:-

```
app:    [S(x,y) → S(x,z) Zero(z,y)]      append zero
app:    [S(x,y) → S(x,z) One(z,y)]       append one
nil:    [S(x,y) → x=y]                   delete 'S'
```

where *x, y, z* are nodes, and '*S*', *Zero* and *One* are dyadic relators. In the rule notation, equality '=' is a relator that identifies two nodes; the need for it emphasizes the fact that we are not rewriting graphically: nodes are not points on a line. These rules are in the form of a Prolog program, with interpretation that:

`S(x,y)` means "node *x* is the start of a string that ends at node *y*".

We see from this that the syntactic description can be expressed in a meta-notation that is not confused with the target notation (that being described). When this is done, the formal grammar is recognizable as kind of proof-system which governs the formation of syntagmatic units from a set of connected lexical units. The formal connection between Prolog programs and hypergraph grammars is also examined in (Corradini *et al.* 1991).

4.4.2 Linguistic Notions of Grammar

What is the relationship between generative and constraining rules in syntax? Although we are not concerned with the issues of natural language, there is reason to believe that linguistic techniques may contain fruitful ideas for resolving this question.

In (§3.1.2) the work of computational linguists was quoted on the subject of semantic grammars. Such work on natural language involves a quest for a combined computational and logical system that would cover the whole range of syntax – and pragmatics also. It is attractive to adopt this approach here in relation to notations. Following Morrill (1993), syntax can be treated as a theory relating graphics and semantics; grammars then take the role of "proof-theoretic meta-theory for the model theory or logic" of the operators on syntactic types.

Our task is thus to investigate the relationship between grammar and logic, and the theme that grammar is concerned with implementing certain kinds of logical inference – the notion of **grammar as logic program.** In fact, the linguistic research gives evidence that processes of 'natural' language are dependent upon tractable logical systems (Dörre *et al.* 1994, Carpenter 1991).

According to (König 1995), grammars for natural languages differ from formal language grammars in that (1) the lexicon contains complex syntactic information, and (2) natural language exhibits non-local syntactic dependencies. It can be seen that diagrams are also similar to natural languages in these two aspects. Lexical items are governed by elaborate connectivity constraints and the arrangement of items is not necessarily determined by strict adjacency conditions.

Some of the techniques of description discussed here were also seen in the last chapter as applied to graphical languages (§3.2). We briefly mention two restricted logical systems that have found favour in recent linguistic theories, in addition to Horn clause logic. These are feature structures and categorial grammars.

4.4.2.1 Feature Structures

Naïvely, features are a way of organizing properties that are appropriate to certain types of item, so that a property of an item is an attribute, presented as a feature-value pair. Features are found in linguistics as basic structured discriminations between items, such as Number, Gender and Person of lexical items. Morrill (1993) explains that features (on basic syntactic categories) simplify reasoning by effecting a universal quantifier as a type-constructor. Feature structures, like Horn clause logic, allow reasoning by *unification* – see surveys of (Knight 1989) and (Sheiber 1986).

Carpenter (1991) develops *typed* feature structures, viewed as an object-oriented generalization of first-order terms, with types arranged in a multiple inheritance hierarchy. In general they may be treated as a special form of constraint logic that can be applied in rule based approaches, as terms for definite-clause logic programming (Aït-Kaci & Nasr 1986). Each (semantic) type must specify the features for which it is appropriate, and the types of values these may take. Features can be ordered according to information content by extending the subsumption order on types, leading to *morphisms* over feature structures that play the role of substitutions for variables. Practically,

adding type information much reduces time spent on expensive structural unification and recursion by early failure detection and the precompilation of taxonomic reasoning.

For Dörre *et al.* (1994), a general feature structure can be viewed as a graph whose edges are labelled with features and nodes carry sets of sort symbols, in which edges leaving the same node must have different labels.

> **Definition**: A feature structure on an alphabet F of features, and an alphabet P of sort symbols, consists of:-
>
> a) a non-empty set D of nodes;
>
> b) for each feature, a partial function on D, that assigns values;
>
> c) for each sort, a subset of D.
>
> Some sorts correspond to unique mutually disjoint atomic structures.
>
> Traditional feature structures may be restricted: e.g. connected, rooted, finite, or acyclic.

They can be given a semantics in FOL over a signature containing all the predicates, leading to *feature constraint languages*. Alternatively, propositional modal logic offers a more concise framework – *feature term languages* – where features are sentential operators and sorts are propositional variables. Feature terms are preferred over first-order terms because subterms are accessed by path-names and terms may be partial since subterms can be omitted (Dörre *et al.* 1994).

4.4.2.2 Categorial Grammar

Categorial grammars are based on a calculus of string concatenation proposed by (Lambek 1958). Dörre *et al.* (1994) refer to the Lambek calculus as an intuitionistic, non-commutative variant of Linear Logic (Girard 1987). Parsing a sentence with the calculus means finding a proof for a goal (the start-symbol) from a database giving the syntactic type of each lexical item in the input string. Lambek (1961) presents a further non-associative version of the calculus.

Multimodal Categorial Grammars (MCG) extend the Lambek calculus to deal with commutative forms and unary modes of combination, needed for other kinds of word-ordering, for encoding syntactic features and to permit copying of resources (Moortgat 1994, Carpenter 1995b). Carpenter provides a sequent-based proof-theoretic presentation of MCG, sufficiently general to accommodate all the various systems that have been proposed – including linear logic with its two modals ? and ! that permit repeated use of premisses.

This proof system is interesting to us because it generalizes the notion of a sequential grammar to admit other relational connectivities, of the kind that might be expected in graph grammars for diagrams.

Feature-Based grammar logics combine feature-unification and categorial grammar and give them a model-theoretic semantics. Dörre *et al.* (1994) apply *fibred semantics* (i.e. layered logics) to augment a Lambek Categorial Grammar (LCG) with Kasper-Rounds logic of feature descriptions (Rounds & Kasper 1986). The two logics are *fibred* together by allowing any formula in one logic (a feature term) to be an atomic formula (a syntactic type) in the other, either way around. A fibring function maps elements (or worlds) of the domain of one logic to elements of the other domain. The rules of both systems are admitted, and further interaction rules may be added. The combined proof-system can be more efficient than a general purpose prover.

They justify the idea by considering Horn-Clause programs. If these are to represent formal grammars, derivations must be restricted: they require *resource awareness*, because all sub-expressions must be accounted for exactly once in a parse. This could be done by using a fragment of linear logic. Alternatively, control arguments be can added: Definite Clause Grammars (DCG) achieve this by encoding an n-place terminal as an (n+2)-place predicate to include start and end string positions.

Lexicalized grammars also combine feature structures with categorial grammars; *LexGram* (König 1995), described as an amalgam of LCG and Head Driven Phrase Structure Grammar (HPSG). HPSG (Pollard & Sag 1994) is based on typed feature terms with an inheritance mechanism. LexGram is derived from HPSG by lexicalizing its Phrase Structure principles and schemata. It extends LCG with the unification formalism CUF (Dörre & Dorna 1993) for handling typed feature terms. HPSG is reduced to one single phrase structure schema which is equivalent to the commutative version of LCG. Word order is treated by adding directional operators. Each HPSG sign then corresponds to a sequent, which has *phonetics* in the antecedent, and the *category* value as its succedent (goal).

Parsing is efficiently restricted via the input word-sequence; top-down and bottom-up parsing can be ideally joined. Modelling is simpler since the only syntactic viewpoint is from the lexemes. The lexicalist approach provides for a uniform view of grammar as a word class hierarchy. It benefits

from a typed-feature-based language: the signature (type system, CUF sorts) has to be defined before the rules, making grammars more transparent. Sorts can then serve as interfaces among grammar modules (König 1995).

We see from these linguistic theories that economy in the logical systems leads to efficiency in parsing, which requires a resource-sensitive proof theory. The two systems can be combined in layers, and it is helpful to establish a signature (a syntactic type system) as a basis for grammatical rules.

4.4.3 Semantic Definitions

In (§3.1.1) it was noted that syntax and semantics overlap each other in linguistics. To complete our examination of syntactic description, we next consider semantic processes. What aspect of the syntactic structure can be called semantic?

On the semantic level of syntax it is harder to fix upon a single notion of structure. Linguistics does not offer much help on this problem. According to Carpenter (1995a), computational linguistics has been "obsessed with the problem of finding the right sort of logical forms" in analysing sentences. He points out that this analysis is both "too hard", since people cannot retrieve *all* semantic information, and "too easy" because it does not take into account all the extra information that people actually take in. In notation we are in a better position, because it is easier to limit the notion of semantics on formal grounds.

In (§4.1.1) the importance of metaphor was emphasized as a feature of linguistic semantics and also of diagrams. Here the connection is clarified and related to work that applies algebraic morphisms.

4.4.3.1 Value in Context

From a denotational perspective, the meaning of an expression is abstracted as a *value* that interacts in some way with its environment. This semantic value has exactly enough structure to explain the effect of enacting the expression in an intended context. It might be seen as a transforming function, or an event in a finite state automaton. In Chapter 3, graph rewriting systems were reported as providing a way of integrating computation with syntactic notational processes (§3.1.3, §3.3.3).

Interpretation can be seen as a process of *calculating* a semantic *value* from a syntactic derivation. These calculations are supported by the syntactic code. Viewers of an expression, on the other hand, do not explicitly calculate meaning; their understanding is instead assisted by analogies – in which metaphor plays a part.

4.4.3.2 Metaphoric and Iconic Processes

Metaphor is a form of rhetoric in which certain words cue a subsidiary domain that is different from the subject domain, thereby forcing the words to take on (temporary) new meanings related to the subject. It is not primarily a graphical phenomenon. In linguistics, the topic of metaphor is not well understood:-

> "I think metaphor is a very difficult problem to say anything concrete about. I do not see how to apply any of the techniques that we know about." Carpenter (1995a)

Carpenter (1995a) wishes to understand how to interpret prepositions like 'in' and 'on', which invoke a spatial analogy. We notice that these concerns have direct counterparts in interpreting spatial relations in diagrams. For example, a topological relation of insideness may act as a cue for the subsidiary domain of 2D space. The spatial relationships in a diagram are then iconically interpreted into the subject domain.

These diagrammatic processes differ from linguistic metaphor in important ways. In diagrams, the cues need not be lexical items, and the induced structure need not be connotational. A graphical cue may suggest a syntagmatic process; in general, metaphors induce coding at some higher level of syntax. In formal notations, the interpretation induced by metaphor is a fixed part of the principal code, not a temporary meaning.

It is evident that interpreting notation requires deductions to be made on the basis of graphical / spatial logic (by whatever cognitive process), but this in itself is not metaphor – it is just recognition and reading. Certainly in some cases diagrams do stimulate visual analysis and spatial thinking: in geometric diagrams and maps, the space on paper represents a different but analogous space. Usually though, the subsidiary domain lies beyond sight and space, in the great variety of experience of the world.

In an explanation of Stenning & Oberlander (1992), Euler's circles denote mechanical linkages, by virtue of which they may connote propositions in syllogistic problems and provide methods of

solution. In such cases, the 'literal' meanings of visual cues are thus aids to learning the code and the required reasoning skills. The code structure supports certain concrete natural or familiar denotations, which by analogy support a formal semantic structure for less understood aspects of the subject domain. By these means, metaphors facilitate reasoning by the specificity of their direct relationship between the subject domain and some subsidiary domain whose deductive chains are familiar.

4.4.3.3 Analogies and Morphisms

How can syntax rules accommodate direct analogies between graphics and semantics?

We do have techniques for describing analogies. The algebraic methods of description reported in the last chapter (§3.2.3) make use of morphisms -- mappings between algebras -- in order to define semantics and metaphor in diagrams. For instance, Wang & Zeevat (1996) describe the analogy between picture and meaning as a partial *signature morphism* from the order-sorted signature of the picture to that of its 'meaning'.

In a similar way, Gurr (1996) discusses analogical characteristics of notations in terms of homomorphisms and isomorphisms between world and representation. He calls the map from world to representation *lucid* if it is injective, and *sound* if it is surjective; the map from representation to world is called *laconic* if it is injective, and *complete* if it is surjective. Thus he notes that, in diagrams, poor abstraction causes a lack of soundness, while the sequential form of texts causes them to be non-laconic. In rigorously defined notations there is found a lack of flexibility, causing incompleteness; users may overcome this by means of *secondary notations* (e.g. informal use of spatial layout to convey information).

How can we make these kinds of analogy uniform across all the expressions of a notation? The complexity of interwoven metaphor and comparative interpretations that holds together the layers of code makes for difficulties in giving a purely grammar-based analysis of graphical notation that can explain its semantics. How can analogy be incorporated into syntactically-based descriptions? The theory developed in the next section aims to answer these questions.

4.4.3.4 An Argument for Logic

The reported research directions in the field of computational linguistics provide an indication that logics and their implementations are at the root of the matter that concerns us. The argument

made here implies that a formally described notation is more that just a defined class of well-formed forms, since there must be also be a logical definition of its semiosis. Generative grammars then take on a role subsidiary to that of axiomatic definitions of structure.

The technicalities of grammar tend to hide the fact that linguistic structure is a solution to the problem of rearranging a possibly multidimensional semantic form into a sequential code. This is a kind of combinatoric packing problem, of quite a different nature from the semiotic problem of finding a conceptual representation for an idea. A discovered derivation for an expression should discard this packing information and provide material for recognition of semantic constructs. The same applies to graphical expressions, except that the two-dimensional medium presents different possibilities and restrictions.

It is proposed that notational packing and parsing aspects also be treated as proof-theoretic operations of some logical system.

> Suppose that we are considering a section of a notation that contains two layers, called graphic and semantic.
> An expression may be regarded as a form that is composed of both graphic and semantic items, and a wff is constrained to be both correct in graphics and semantics, and to satisfy the rules that coordinate them. By specifying a grammar, we are providing:
>
> (1) a guide for expressing any semantic form in the particular graphics, and
> (2) a means of (uniquely) inferring the semantic layer of any wff from its graphic layer alone.

This method of integrating the structural levels calls for the flexibility of approach afforded within a general framework of descriptive logic. A grammatical rule-system furnishes *proof-theoretics* that can be seen as implementations for constructive fragments of the logic. For the most complex of the semantic or pragmatic levels, involving translation, reasoning and calculation, a 'grammar' is the same thing as an abstract program.

Though there are many logical systems available for general purposes in mathematics, the problem remains to select the logic that best suits the whole domain of graphical notations. Restricted fragments of this logic would then be appropriate for different semiotic layers. The perspective argued for could be called *meta*logical, in that the logic chosen to describe semiosis would not be tied to any specific topic of computer science or theory of software development. Neither would it depend on the graphical (pictorial and spatial) properties of a particular notation.

4.4.4 The Form of Expressions

To complete this section, we wish to attend to the problem of defining the underlying form that expressions of a notation exhibit. If all expressions of a notation belong to a wider class *F* of forms that can be manipulated in grammatical and other processes, how can we characterize the whole of *F*? What kinds of internally structured object would qualify as a constructed form?

Answering these questions will require recourse to the ideas and terminology of Category Theory, which however will not be explained at this point. The reader who does not have any familiarity with categories may want to re-read this passage after absorbing the next chapter.

To pursue an informal argument about structure, operations on parts will here be used to derive a notion of forms as general 'graphoid' constructions. Since we wish to cover many cases in an uniform manner, this will be done by a process of generalizing and completing: i.e. filling in structured spaces by closing under all operations and admitting all provable properties. This method is preferred in mathematics because it leads to simple tidy structures.[13]

4.4.4.1 The Causes of Structure

Why should expressions be regarded as structured? How does structure arise? The word 'structure' means something that is *built*. In order to build something, there must be parts and pieces to build it from, and building techniques or operations. This suggests a way to resolve the problem.

Received wisdom [in computer science] holds that the structure of an object depends on how one intends to operate with it. Accordingly, we seek to define a form-space *F* as a class generated by constructive operations. Structure will therefore not be based on hypothesized mental models.

If expressions were merely representations of abstract data, we could define their structure according to the operations used to construct them. An expression might be resolved as a sequence of primitive drawing or editing events. This leads to a very cumbersome representation, since the event-sequences are unbounded, and we have to determine when two sequences are equivalent, i.e. taken as building the same component.

More flexibly, we might construct the expression from primitives in an hierarchical manner, so that

---

[13]These structures are often infinite. By contrast, in computing it is more common to have partial operations, owing to resource limitations of many kinds.

at any stage a new part is composed from parts made in previous stages. The result can be seen as an algebraic term, by naming each constructor operator and defining its *arity* as a sequence of *sorts* of parts that it may combine. The term-data represent the expression as a tree-structure whose nodes are labelled by constructor-names and edges are labelled by sorts. This tree may also be regarded as a hypergraph whose edges are constructors. Once again there will be many equivalent trees representing the same expression.

String and term rewriting could be used to calculate equivalences – but if the constructors do not relate to syntactic processes, these methods only introduce irrelevant complexity. What we seek is a representation that deals with syntactic types more directly.

4.4.4.2 Graph Representations

Graphs offer a direct approach to representing structure, in which *incidence* relations specify how a few different kinds of elements connect. Current approaches to diagram syntax, as reviewed in the previous chapter, often describe expressions in terms of graphs or relational structures. Justification for this method is given (if at all) on the empirical grounds that the graphs were found to work for the cases considered. Unfortunately there is little agreement on what type of graph to employ. Although a wide variety of graphtypes exist, different types sometimes prove to be equivalent in structure. No particular class of graphs stands out as the best candidate for uniformly defining notation structure.

The choice of type of graph may imply some corresponding cognitive basis for diagram structure. For example McWhirter (thesis: 1995) justifies his own use of *relational structures* as a modelling notion by an assumption (*ibid.* p29) that infants experience the world in terms of 'things and relationships', a world view which graphs formally reflect. He adduces no evidence for this hypothesis; it is hard to see how anyone could do so. We would prefer a sounder reason for the choice.

4.4.4.3 Parts of a Whole

This issue is resolved as follows. The source of much structure is the practice of analysing expressions into parts; it concerns the part-whole relationship, as can be observed in the particular example of text. The processing and interpretation of an expression is then established upon this analysis. There is a desire to find smallest parts and largest parts, to have a way of building large

parts from small parts, and to allow rewriting: the replacement of a part in an expression with a new part, leaving other parts unchanged.

We first observe some elementary properties of the part-whole relationship by reference to a textual example. This leads to the contention that forms belong to a *category*.[14]

> Consider for instance, strings based on the Roman alphabet. Then the string `abracadabra` has parts
>
> `a,b,c,d,r, ab, br, ra, abra,`
>
> and many others. Some parts occur severally in different places.

To cope with multiple occurrence, it is convenient to consider tokens rather than abstract parts in expressions. We may observe the following definitions and rules:-

D1) An *occurrence* is a relation p of containment between two tokens:

`P = part(p), Q = whole(p);`

that is to say, P is copied as a part of Q. This is written:-

$p: P \to Q$

R1) Every token P occurs as itself; we identify P with its self-occurrence:

$P: P \to P$

An important property of occurrence is that it is transitive: if P occurs in Q and Q occurs in R, then P occurs in R.

D2) Two occurrences p, q are said to be *compatible* if `whole(p) = part(q).`

R2) If p, q are compatible occurrences, there exists a composed part `p;q` such that:

`whole(p;q) = whole(q)` $\wedge$ `part(p;q) = part(p).`

In symbols, this can be expressed:

$$\frac{p: P \to Q \qquad q: Q \to R}{p;q : P \to R} \text{ (R2)}$$

This structure is called a *deductive system* in (Lambek & Scott 1986), because the occurrences behave like proofs of propositions, and the rules deduce proofs. We observe further, in the case of text, that certain compound parts may be equated:

The operation of composition of occurrences has left and right identities, and is associative:-

---

[14] It is unfortunate that the words 'category', 'categorial' and 'categorical' each have different technical meanings in linguistics, logic and mathematics – as well as informal meanings. From here on, unless the context dictates otherwise, the mathematical usage may be assumed.

```
P;p = p = p;Q                (E1)
```

If *p* compatible with *q* and *q* compatible with *r*, then

```
(p;q);r = p;(q;r)            (E2)
```

A structure with these properties is called a *category*. It follows that if we wish to refer to parts and wholes in the same manner as for text, we are obliged to treat tokens as *objects* in a category whose *arrows* are occurrences. In addition, by extending this category with more objects, arrows and properties, we can admit many ways of manipulating non-textual expressions.

4.4.4.4 Joining and Operating on Parts

We may consider expressions as being built with the help of forms which need not satisfy the desired syntax – e.g. parts of expression-tokens. Maps can be seen as operations that send any part of one form *P* to a corresponding part of another, *Q*; they show where *P* occurs as 'part' of *Q* in a more general sense, because maps need not be 1-to-1. Within this wider setting, simple parts are *subforms*, seen as 1-to-1 maps in which P is isomorphic to some part of *Q*.

These considerations suggest a certain way of completing the category. Technically speaking, a general category *F* of forms can then be characterized from standard theory (MacLane 1971) as follows:-

> The *objects* in *F* are forms. Its *arrows* are general maps from a form *P* to a form *Q* that define how the (possibly overlapping) parts of *P* map onto the parts of *Q*. This is achieved by allowing certain *limits* and *colimits* to be constructed:-

To build an expression, we may wish for example to join forms together, or to find the largest part in which two subforms overlap.

> Joining requires the apparatus of finite colimits (e.g. *pushouts*).
>
> We may need a notional empty form -- an *initial* object in *F*.
>
> Subforms are the *monic* arrows in *F*. Finding overlap of subforms will require some finite limits (*pullbacks*).

If we are generous enough to admit all such (limit and co-limit) constructions – which may not be necessary – the result will be a finitely *bicomplete* category. This will more than suffice for rewritings of forms based on the double-pushout approach of graph grammars mentioned in (§3.1.3).

One aspect of this generosity is that we can construct a form which is the *product* of two forms *P*

and Q, say, in the following sense: a map to the form `P × Q` represents two maps, one to *P* and one to Q. A map from the product to *R* in effect assigns any pair `(p,q)` of parts to a part of *R*. For instance consider a discrete version of 2D Cartesian geometry, as might divide a rectangle into pixels. Take *P* and Q as sequential forms that model sections of the X-axis and Y-axis respectively. The form that holds the 2D space defined by these is the product `P × Q` of the two axes; a subform of this product would provide a representation of a simple drawn shape – a 'bitmap'.

A map may also be seen as way of labelling or classifying parts. Taking the previous example, we might take a form C denoting a colour-space; a map `p: P × Q → C` is a colouring of the rectangle. Cases like these show that there is some benefit from extending operation on forms beyond mere joining and deleting of parts of expressions, into much richer constructions.

The result of this logical completion is that the range of admissible types of syntactic part has been extended to cover all combinations that can be 'imagined' or construed, rather than just those that are clearly indicated in some syntax.

4.4.4.5 Representing Properties of Parts

This argument does not take us much further. The next step is to build *computational* structure into our categories of forms, by introducing notions from Set Theory. This construction results in a kind of category known as a *topos*: a category that embodies an intuitionistic type theory, as fully described by Lambek & Scott (1986). Basing descriptions of notation processing on a topos *F* of forms allows us to define rewriting operations in abstract, without recourse to encodings in set theory. By such means we can for instance encode general rewrite-rules in the manner of Bauderon (1996), who uses a method of labelling to classify which parts of a graph are to be deleted in a rewriting.

Topos Theory gives us a notion of logic that does not rest on manipulations of formulae.

> In a topos, facts about parts are represented as *equations* between maps. Equations enable us to reason about forms without needing a syntactic encoding of them. This is done by constructing forms that represent arrangements of other forms, and regarding them as *types*. Maps are viewed as *terms* that denote operations on types – or indeed functional programs. *Propositions* are terms of a certain type.

First we note that any complete category has an object that is an individual – which combines all its parts into a whole.

Engels, G.; Lewerentz, C.; Schaeffer, W. (1987) Graph Grammar Engineering: A Software Specification Method; in (Ehrig et al. 1987) LNCS 291 p186-201.

Ferrucci, F.; Tortora, G.; Tucci, M. & Vitiello, G. (1994) A Predictive Parser for Visual Languages Specified by Relation Grammars. IEEE Symposium on Visual Languages, 1994 p245-252.

Ferrucci, F.; Tortora, G.; Tucci, M. & Vitiello, G. (1996) On the Generation and Recognition of Visual Languages: Relation Grammars and Related Approaches. In: TVL'96: International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Ferrucci, F.; Pacini, G.; Satta, G.; Sessa, M.; Tortora, G.; Tucci, M.; & Vitiello, G. (1996), Symbol-Relation Grammars: A Formalism for Graphical Languages. Submitted for publication.

Freyd, Peter (1972) Aspects of Topoi. Bulletin of the Australian Mathematical Society 7. p1-72.

Galton, A. (1988) Formal Semantics: Is it Relevant to Artificial Intelligence; AI Reviews 2(3) 1988

Gee, David M. (1995) Dept. of Computing, University of Nothumbria at Newcastle.

Gehani, Narain (1985) Specifications: Formal and Informal -- A Case Study; in (Gehani & McGettrick 1985) p173.

Girard, J.-Y. (1987) Linear Logic. Theoretical Computer Science 50, p1-102.

Godwin, W.H. (1991) Some Proposals Towards a Theory of Notation in Software Engineering. In De Neuman, Bernard et al. eds. (1991) Mathematical Structures for Software Engineering: IMA Conf. series 27, Oxford University Press, p53-82.

Goel, Vinod. (1992) "Ill-Structured Diagrams" for Ill-Structured Problems. In Proc. AAAI Symposium on Diagrammatic Reasoning, Stanford University, March 1992 p66-71.

Goguen, Joseph A. (1985) More Thoughts on Specification and Verification; in (Gehani & McGettrick 1985) p47.

Goguen, J.A. (1988) What is Unification?; Report SRI-CSL-88-2R2. Also in Nivat, M. & Aït-Kaci, H. (eds.) (1989) Resolution of Equations in Algebraic Structures, Vol. 1: Algebraic techniques. London: Academic Press, p217-261.

Goguen, J.A. (1997) Semiotic Morphisms. (Draft paper) Dept. of Computer Science and Engineering, University of California at San Diego.

Goguen, J.A & Burstall, R.M. (1984) Introducing Institutions. LNCS 164, Springer, Berlin.

Goguen, Joseph A. & Burstall, R.M. (1986) A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments. In (Pitt et al. 1986) LNCS 240 p313-333.

Goguen, J.A & Burstall, R.M. (1992) Institutions: Abstract Model Theory for specification and programming. Journal of ACM 39(1) p95-146.

Goguen, J.A. & Malcolm, G. (1997) A Hidden Agenda. Technical Report CS97-538, Dept. of Computer Science and Engineering, UCSD.

Goguen, J.A. & Meseguer, J. (1987) Models and Equality for Logic Programming. In Ehrig, H.; Kowalski, R. et al. eds (1987) Tapsoft'87 vol2, LNCS 250; Springer, p1-21.

Goguen, J. & Meseguer, J. (1989) Order-Sorted Algebra 1: Equational Deduction for Multiple Inheritance, Polymorphism, Overloading and Partial Operations. Tech. Report SRI-CSL-89-10, SRI International.

Goldblatt, R. (1986) Topoi: Categorical Analysis of Logic; Studies in Logic and Foundations of Mathematics 98, North Holland 1979 (2nd Edn. 1986).

forms by arrows in the topos.

Although we cannot justify the whole of topos structure as fully essential to our purposes, we would not gain anything by restricting *F* to fewer constructions. This kind of structural description seems most economical because it is equivalent to a simple intuitionistic type theory. The claim is not that forms are *in essence* graphoid in shape, but that this approach to defining them is adequate and not too general.

4.5 Towards a Theory of Notation

The remaining task of this chapter is to show how formal support can be given for a theory of notation. We consider how the structural notions arising from the above investigations can uphold the pragmatics of formal notation. As a first step to resolving the problems of notation description and usage, a formal theoretical foundation for semiosis is put forward.

4.5.1 Semiosis in Notations

The exploration of the previous section has concentrated upon the syntactic structure of notation. Here the whole semiotic function is outlined, including the pragmatics of notation in a software development context. It is argued that every notation embodies a practical method of reasoning – an instructive logic. Together, notations provide grounding for the application of mathematics to a wealth of problem areas. Our focus in what follows is less upon description of a natural phenomenon, and more upon how we might support the design of notations to suit specific usage.

4.5.1.1 Summary of Code Structure

Graphical expressiveness is maintained by levels of structure that bridge the gap between physical medium and subject domain. Following a linguistic model, these can be divided into four layers:-

Semantic	– deduced conceptual structure
Tagmatic	– grammatical arrangements; simple recursion
Lexic	– pictograms, links, enclosures, frames
Graphic	– geometric elements, colour, texture, style

The *graphic and lexic* layers are two articulatory levels, allowing complex shapes and relations to be built from elementary parts. The *tagmatic*[15] *and semantic* layers concern various 'linguistic'

[15]Or syntagmatic – from Greek ταγμα : something arranged or ordered.

patterning or analogical devices that supporting denotation and connotation. The number of layers may vary, and the division is somewhat arbitrary. The logical complexity of each syntactic layer is limited by certain cognitive resource constraints, dependent upon the purpose of the notation and the skill required of users. It is these limitations that determine the character of the syntactic coding – analysed as constructive logical relations between semiotic layers.

The structural parts of expressions will be called *items*; they generally belong in the 'semiotic hierarchy' defined by the layering. By layers, they may be termed *graphemes*, *lexemes*, *tagmemes* and *sememes*; these correspond respectively to characters, words, phrases and meanings in narrative text.

Each level has a *combinatoric form*: an underlying form that captures each expression as a 'graphoid' configuration of items of various sorts. These abstract expression-forms are said to be *embodied* or realized (e.g. geometrically) in the medium, while they are *interpreted* in the subject domain, providing interfaces with both the physical world and the cultural world of shared ideas. Embodiment of graphics is concerned with universal spatial, perceptual and cognitive properties of the medium. Some formalization of these properties, independent of choice of notation, is needed to bind the graphical layer to the ground of physical nature. The system of interpretants of the signs in an expression form the semantic interpretation in the subject domain. Where there is a connotational semantics, the interpretants are themselves vehicles that signify in higher semantic levels (see §4.2.1 above).

Why have layers? In the previous chapter (§3.4.2) it was suggested that design of notations gains flexibility from a separation into layers, for practical reasons that do not depend on their empirical psychological existence. The reasons are twofold: such a modular construction has the benefit of making syntax simpler to modify, and each layer (or module) may have different logical complexity, as has been discussed above. In addition, freedom to change the lower levels makes it possible to design in the visual or linguistic metaphors that help the user to learn and reason with the notation.

4.5.1.2 From Graphics to Pragmatics

We seek ways to accommodate these layers and to connect expressions within their intended environment. In pragmatic terms, when notations are used in a software development setting,

expressions of all kinds are displayed in some physical medium and *enacted*: communicated to (or by) some computational system. A formal pragmatics is possible only when enactment occurs in definable situations that conform to specified presuppositional constraints.

> Whereas signs within expressions have an abstract *interpretant*, replicated tokens within displays may be associated with a specific *referent*: a concrete object or state that is to be found in the whole context that surrounds its display. Deictic tokens in an expression-occurrence must point to appropriate items in the situation, for example.[16]

This pragmatic 'interface' can thus be dealt with by formalizing the context, independently of the notation, as a complex of well-formed situations. Unlike denotational semantics, this abstracts the situation of expression-occurrence and its tokens along with the situations referred to by the expression. We are concerned with a *situated expression*: an expression embedded in a formal presupposed context.

The expressive medium and its enclosing context can carry structures well beyond the size and complexity restrictions appropriate to individual cognition. Computationally, this might be simulated by a constraint logic or rewriting system that co-ordinates the propagation of changes within the whole context.

4.5.1.3 A Community of Notations in Context

What kind of notational edifice for software development could we hope to build on the strength of a semiotic theory? In the survey (§2.2) of Chapter 2 we saw that a software development project is a complex 'animal' with many aspects, requiring many notations. For a realistic approach, we could define a *community* of notations as a set of related *logical instruments* within a common environment. In each notation, we then envisage a contextual well-formed form as a situated expression – an expression augmented by some segment of the formal context, and bound by pragmatic constraints that determine what forms may occur in a particular kind of context, and how the environment might be expected to behave as a result. This would for instance cope with expressions within a system specification document, or to diagrams that propose a change to a system-design representation during development, or to diagrammatic commands issued by the

---

[16]The Hyperproof system – a computer program to help students learn to reason using both pictures and of first-order logic (Barwise & Etchemendy 1988, 1994) – employs this kind of analysis. A displayed logical statement refers to a situation that is pictured on screen.

end-user to be executed by an application.

A document can be treated broadly as an arrangement of expressions in several notations, referring to a specific context: a complex of situations. Documents are subject to modification by editing, translation and deduction. In dialogue and other discourse processes, changes occur in a document as a result of interactions with agents in the context. The propagation of change is subject to logical constraints defined on the document structure and each of its notations.

The reviews in (§3.3) suggest that a notation-community might be based around a *core structural plan* of the project environment, a compound semantic domain, upon which each notation holds its own viewpoint. An expression is a selected *view* of a possible context or single situation. The core project database is never notated in its totality, but provides a semantics for all internal checking, evaluation and proof, and a route for all translation.

4.5.1:4 Notation as a Logical Instrument

Another aspect of pragmatics is the understanding and skills of participants. The analysis of notational roles in (§4.1.3) found that participants use notations for thinking with or for communicating thoughts, more or less formally, in software development. It was asserted in (§4.1.1) that notations are a way of bringing mathematics and logic to bear on problems. Here we discuss how this occurs.

A formalist viewpoint posits that all explicit mathematical concepts can be studied without going beyond the formal properties and transformations of their notations.[17] The fact that expressions are finite and never very large avoids any paradoxes connected with infinity, and the question of computational complexity becomes a matter of notational economy. This suggests a concept of *semiotic mathematics*, allied to the constructive theories that are motivated in computer science logic, where symbol manipulation is the only option available for problem-solving. Vaughan Pratt (1988) makes some interesting points in this regard.

---

[17] A personal realization of this perspective was the author's original inspiration for this endeavour of pursuing notation research.

"... it is a tenet of faith that 'conventional' mathematical proofs can be expanded out to a purely set-theoretic argument, yet this is almost never done. Moreover category theory has in recent years posed a challenge to set theory as an alternative and strikingly different foundation, indicating the non-uniqueness of such expansions. The possibility then arises that no such foundation is needed. Instead we may consider any given argument as being conducted in one or more relatively small and localized theories."

"I propose that the proper notions of constructivity in a logic are its computational complexity and its human surveyability... This then speaks for computational tractability as an important criterion for judging the merits of any theory."

We may surmise that notations are presentations of the 'relatively small and localized' constructive theories referred to, made visible for human survey and formalized for tractability. This position is supported by the cognitive theories advanced by other authors. Stenning (1994) notes the "strong vein" of graphical thinking in the development of logic and thus he refutes the assumption that logic is sentential – it is rather "an abstract consequence relation which can be implemented in many mechanisms." The implementation should be made explicit. He sees diagrams as weakly expressive systems that are cognitively useful when their power is sufficient to the task at hand, where they offer *inferential tractability*, but that are pathological when abstraction is required.

4.5.1.5 Tractable Reasoning with Notation

Following the work of Levesque (1988), Stenning & Oberlander (1992) argue that FOL cannot provide a computationally tractable reasoning system. Levesque claims that the modifications to classical logic found necessary to ensure tractability are exactly the same as are necessary to make logic psychologically realistic; a primary reason for the appeal of visual information lies in what it cannot leave unsaid about the observed situation – its *vividness*.

> For a sentence in FOL to be vivid it can only contain ground, function-free atomic sentences; unique names; universal sentences over a closed world; and the axioms of equality... A vivid knowledge base "looks like its described subject matter" (Levesque 1988).

Accordingly, Stenning & Oberlander take a model-theoretic perspective, and ask how *many* models correspond to an expression (when viewed as a proposition). To improve tractability, it is necessary to minimise the number of cases to be considered. Levesque suggests ways to increase expressiveness, such as the use of Horn clauses and semi-Horn forms to encode taxonomies, allowing some disjunctions to be hidden in subsuming predicates – or the use of unsound reasoning. He notes that "Observer-centred visually salient properties become defaults"

"... it is a tenet of faith that 'conventional' mathematical proofs can be expanded out to a purely set-theoretic argument, yet this is almost never done. Moreover category theory has in recent years posed a challenge to set theory as an alternative and strikingly different foundation, indicating the non-uniqueness of such expansions. The possibility then arises that no such foundation is needed. Instead we may consider any given argument as being conducted in one or more relatively small and localized theories."

"I propose that the proper notions of constructivity in a logic are its computational complexity and its human surveyability... This then speaks for computational tractability as an important criterion for judging the merits of any theory."

We may surmise that notations are presentations of the 'relatively small and localized' constructive theories referred to, made visible for human survey and formalized for tractability. This position is supported by the cognitive theories advanced by other authors. Stenning (1994) notes the "strong vein" of graphical thinking in the development of logic and thus he refutes the assumption that logic is sentential – it is rather "an abstract consequence relation which can be implemented in many mechanisms." The implementation should be made explicit. He sees diagrams as weakly expressive systems that are cognitively useful when their power is sufficient to the task at hand, where they offer *inferential tractability*, but that are pathological when abstraction is required.

4.5.1.5 Tractable Reasoning with Notation

Following the work of Levesque (1988), Stenning & Oberlander (1992) argue that FOL cannot provide a computationally tractable reasoning system. Levesque claims that the modifications to classical logic found necessary to ensure tractability are exactly the same as are necessary to make logic psychologically realistic; a primary reason for the appeal of visual information lies in what it cannot leave unsaid about the observed situation – its *vividness*.

> For a sentence in FOL to be vivid it can only contain ground, function-free atomic sentences; unique names; universal sentences over a closed world; and the axioms of equality... A vivid knowledge base "looks like its described subject matter" (Levesque 1988).

Accordingly, Stenning & Oberlander take a model-theoretic perspective, and ask how *many* models correspond to an expression (when viewed as a proposition). To improve tractability, it is necessary to minimise the number of cases to be considered. Levesque suggests ways to increase expressiveness, such as the use of Horn clauses and semi-Horn forms to encode taxonomies, allowing some disjunctions to be hidden in subsuming predicates – or the use of unsound reasoning. He notes that "Observer-centred visually salient properties become defaults"

(e.g. apparent right angles in a geometric figure are assumed to be formally constrained unless otherwise marked).

Stenning & Oberlander regard a *proposition* as a logical concept independent of any representation system. They therefore reject the well-known "Imagery debate" – on whether mental images are encoded analogically or propositionally – as being a source of confusion. To reduce the burden on working memory in such tasks, humans may exploit a set of special purpose cognitive mechanisms, developed for perceiving and reasoning about the spatial world, for reasoning about other domains. A reasoning task that can be done with a restricted logic can take advantage of an implementation that would be impossible for more general logics.

In view of the discussion in (§4.4.3), we may surmise that reasoning is very dependent upon the skill of applying metaphors, which in mathematics and elsewhere are often standardized and encoded in notations.

4.5.2 A Foundation: Notation Tectonics[18]

The task of this work is not so much to formalize notations as explicitly implemented logical instruments, in the above sense. Rather, the intent is to lend support to this notion by making explicit the inherent logic of a layered semiosis. Here a basis is proposed that affords both model-theoretic and proof-theoretic stances, as suggested by (§4.4.3).

4.5.2.1 A Metalogical Approach to Notation Specification

We wish to defined formally the connectivity properties of the graphical form of expressions and relate them to the structure of the subject domain. We must formulate syntax in a way that is independent of any logical framework designed for the subject system, otherwise we would have to change descriptive techniques every time a new subject was chosen. In order for descriptions to cover a wide range of system structure uniformly, they must be based on a *metalogic* which can encompass *theories* about such semiotic structure.

The proposed approach is to build **theories** of syntax for subject notations. Minas & Viehstaedt (1995) introduce the term *diagram class* to refer to a notation, or specifically its set of well-formed

---

[18] Tectonic: of building or construction.Tectonics: whole art of producing useful or beautiful buildings; structural features as a whole. (Concise Oxford Dictionary).

diagrams. In our case this will be the class of **models** for a syntactic theory. Propositions in a theory act as constraint rules that can test whether a form is well-formed as an expression: I.e. whether it is an acceptable *model* for the theory. Maps between theories give us ways to refer to the translating, encoding or instantiating of semiotic structure.

The semantics of a notation is regarded as an abstraction of various situations within its subject context, defined by means of a logical theory. Spatial and cognitive constraints determine how such a logically defined structure can be notated as a drawing. A notation's coding builds a relationship between a theory *S* in the semantic domain and a theory *D* of drawings. In the simplest case there is a coding of *S* in terms of *D*.

> Consider first a coding `T: S → D` that expresses a semantics *S* in a graphics *D*. If *p* is a picture, i.e. a model of *D*, then `T(p)` is a model of *S*: a corresponding formal meaning. Expressions in the medium that are governed by *D* can then be interpreted uniquely in semantics, with all semantic properties determined from graphical ones. (Note that interpretation is in the opposite direction to the theory mapping.) Two different pictures *p* and *q* have the same meaning under *T* when `T(p) = T(q)`.

In order for this to work, the theory *D* must incorporate all of the conventions that determine the encoding. In practice we are likely to have a limited theory *G* that establishes the perceptual conventions of the medium, to which semiotic structure must be added. It must be stressed that *G* here is not intended as an absolute theory of drawings, but as an encoding of the natural and cultural expectations about drawings that are appropriate to the notation and circumstance. The arrow here can be taken as informally representing the cognitive effort involved in viewing and interpreting expressions, or some equivalent computational cost.

4.5.2.2 Syntactic and Pragmatic Relations

In the general case some pictorial forms will be without meaning, while some subject concepts will be inexpressible in the notations. To treat this case we consider the semiotic relation that determines which features of drawings are significant and what contextual features they stand for, as a pair of maps spanning the semantical and graphical theories. The first map determines the part of the subject domain that is represented, and the second selects the salient part of the expressive domain: the medium. (For the moment we do not specify the nature of these maps.)

> Let theories *G* and *S* be related by a *span* – a theory *R* equipped with two maps:
>
> `G ← R → S`

diagrams. In our case this will be the class of **models** for a syntactic theory. Propositions in a theory act as constraint rules that can test whether a form is well-formed as an expression: i.e. whether it is an acceptable *model* for the theory. Maps between theories give us ways to refer to the translating, encoding or instantiating of semiotic structure.

The semantics of a notation is regarded as an abstraction of various situations within its subject context, defined by means of a logical theory. Spatial and cognitive constraints determine how such a logically defined structure can be notated as a drawing. A notation's coding builds a relationship between a theory *S* in the semantic domain and a theory *D* of drawings. In the simplest case there is a coding of *S* in terms of *D*.

> Consider first a coding `T: S → D` that expresses a semantics *S* in a graphics *D*. If *p* is a picture, i.e. a model of *D*, then `T(p)` is a model of *S*: a corresponding formal meaning. Expressions in the medium that are governed by *D* can then be interpreted uniquely in semantics, with all semantic properties determined from graphical ones. (Note that interpretation is in the opposite direction to the theory mapping.) Two different pictures *p* and *q* have the same meaning under *T* when `T(p) = T(q)`.

In order for this to work, the theory *D* must incorporate all of the conventions that determine the encoding. In practice we are likely to have a limited theory *G* that establishes the perceptual conventions of the medium, to which semiotic structure must be added. It must be stressed that *G* here is not intended as an absolute theory of drawings, but as an encoding of the natural and cultural expectations about drawings that are appropriate to the notation and circumstance. The arrow here can be taken as informally representing the cognitive effort involved in viewing and interpreting expressions, or some equivalent computational cost.

4.5.2.2 Syntactic and Pragmatic Relations

In the general case some pictorial forms will be without meaning, while some subject concepts will be inexpressible in the notations. To treat this case we consider the semiotic relation that determines which features of drawings are significant and what contextual features they stand for, as a pair of maps spanning the semantical and graphical theories. The first map determines the part of the subject domain that is represented, and the second selects the salient part of the expressive domain: the medium. (For the moment we do not specify the nature of these maps.)

> Let theories *G* and *S* be related by a *span* – a theory *R* equipped with two maps:
>
> `G ← R → S`

The map to *G* ignores items that do not contribute to meanings; it also serves to carry graphical constraints that become *intrinsic* to the syntax. These are mapped into corresponding constraints of *S*. Any properties of *G* that are ignored will become restrictions on expressiveness.

The map to *S* adds constraints from the subject domain, which become part of the *extrinsic* syntax.

Here the two maps also show to what extent the semantics is related to conventional graphical properties – the extent of analogy. Whether this analogy is graphical or syntactic depends on how complex or direct is the map to *G*. The properties of the medium may restrict the range of meanings that can be directly expressed, especially if the map to *G* is unwisely chosen.

The process of interpreting a drawing (model of *G*) involves translation by analogy and syntax to an abstract expression (model of *R*), followed by the construction of a compatible semantic situation (model of *S*), if possible – i.e. if the expression is well-formed. In a formal circumstance, we would want the map to *S* to have at least a partial inverse, so that an unambiguous meaning could be found for a subset of drawings. Otherwise the viewer's informal knowledge of specific context and what is feasible may help resolve any ambiguity.

Within this paradigm we can also describe the pragmatics. The indirect relation between semantics and graphics can be defined by a situated theory that constrains the occurrence of an expression, within a context which will include the referred-to situation. This theory encompasses concepts of both semantics and graphics, while respecting the analogy.

Theories *G* and *S* can combine within a situated theory *K* of expressions-in-context, which enriches them with further concepts and constraints.

$$G \rightarrow K \leftarrow S$$

The advantage of this scheme is that we can go beyond a simple abstract comparison of drawing and meaning. In case a drawing contains deictic references, we are able to include in *K* concepts of drawings as physical objects in proximity to their referents. Here we can think of the process of interpreting an ambiguous drawing as an attempt to construct a compatible context (model of *K*), which is then translated to a semantic situation. This translation will ignore all aspects of the context not connected to the meaning of an expression.

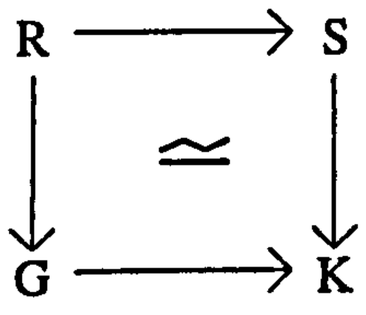


Figure 4.1 Analogy as isomorphism

Taking these semiotic relations together, we see that every concept in *R* has two corresponding items situated in *K*, one in a drawing and the other in the referred-to situation. These two images of *R* are isomorphic – which is indicated by the diagram in [fig 4.1]. This then demonstrates a general framework that explains analogy in terms of a *natural* family of isomorphisms between drawings and meanings.[19]

Full expression of a subject will typically require several notations. A 'core structure' *S* for a diverse subject area can be covered by a community of notations:

$$\{i:I \mid G \leftarrow R_i \rightarrow S\}$$

which represent different (overlapping) views of the domain *S*.

4.5.2.3 Designing Layers of Structure

We may now expand this to a layered framework that relates graphical, lexical, syntactic and semantic steps of a notation *M*, in [fig. 4.2].

Suppose we have selected for notating, a domain whose conceptual structure is represented by a theory *Semantic* which is a part (subtheory) of some larger subject domain. We wish to express its ideas in the medium of drawing, for which we have a standard theory (*Medium*). Given a well-formed drawing, we want to be able to interpret its meaning by a logical construction in *Semantic*. Each semantic part would be defined in terms of a part of a drawing.

Ideally we want translations in *M* via the layers from *Semantic*: the theory for part of the subject domain, to *Graphic*: a theory of drawn forms in the medium:

$$\text{(Medium)} \leftarrow \text{Graphic} \leftarrow\!\sim \text{Lexic} \leftarrow\!\sim \text{Tagmatic} \leftarrow\!\sim \text{Semantic} \leftarrow \text{(Subject)}$$

where the translation is the composition of three maps, which will usually not be total, between separate theories in *M* for each layer. This will be approached from the more general case, which describes a notation that is being designed, and is not yet complete – or perhaps is evolving.

$$\text{Graphic} \leftarrow \text{GrLx} \rightarrow \text{Lexic} \leftarrow \text{LxSn} \rightarrow \text{Tagmatic} \leftarrow \text{SnSm} \rightarrow \text{Semantic}$$

---

[19] Informally, [fig 4.1] can be taken as a depiction of the compatibility between the cognitive association *R* of drawing to meaning and the pattern of occurrences *K* in different contexts.

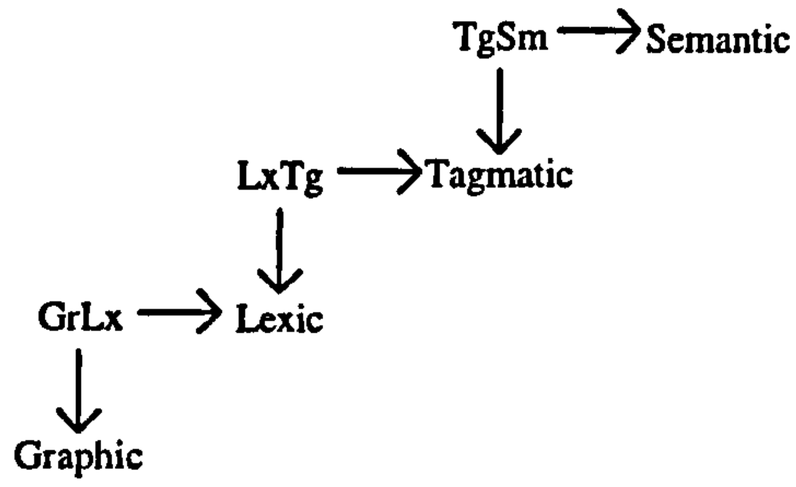


Figure 4.2 Steps between layers

The spanning theories *GrLx, LxSn* and *SnSm* and their maps are *steps* connecting the layers, and indicate analogies which assist the interpretation. Upper steps normally involve more complex operations than lower.

The steps in [fig 4.2] may admit superfluity, ambiguity and inexpressiveness at each step. To make the steps effective, each vertical arrow must select salient items and analogies from its lower layer to its upper, while the horizontal arrow of each step may add further constraints to the well-formedness criteria. The figure also allows for extraneous items to be added along each horizontal arrow, admitting vagueness: (the undesirable possibility of) unnotated determinants of meaning such as tacit context.

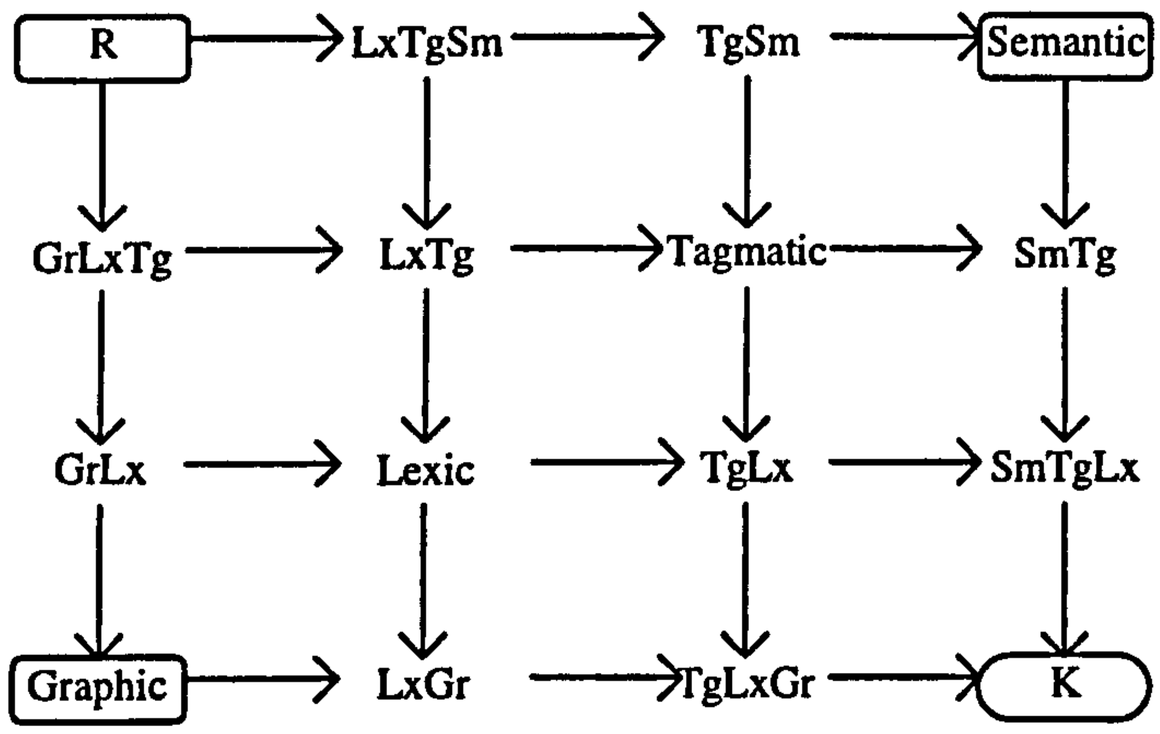


Figure 4.3 A fabric of layers

Above the steps [fig. 4.3] we can construct the largest theory that connects graphics to semantics via analogies. Underneath these steps are formed the combined theory *K* , the smallest that contains all and only the items from all layers of *M*, and into which both Graphic and Semantic are mapped. Ambiguity is removed when the horizontal maps have partial inverses.

The figure factorizes the simpler picture of [fig 4.1] with the proviso that *R* is maximal and *K* minimal. Then all chains of translations that join *R* to *K* are equivalent.

4.5.2.4 Theory Morphisms as Translators

How can we formalize these maps? For this we again require Category Theory. Following the methods of Lawvere we will view theories as *categories* with extra closure properties that define what *doctrine* of logic is to be used. Morphisms between theories (which may be called translators) are required to be *functors* that preserve these doctrinal properties. Models of theory *T* are morphisms from a theory-category to a ground category *Z* – such as the category of Sets and functions. The objects of a theory then become types of item in a model; the arrows of the theory become maps between types, i.e. kinds of association.

These translations are implemented somehow by methods of construction and deduction, of varying complexity.

4.5.2.5 Interpreting Pictures

Under this formulation, interpreting pictures becomes a matter of composing morphisms. Given a theory *R* spanning graphic and semantic theories, we have two translations:

```
rg: R → Graphic
rs: R → Semantic
```

These morphisms describe the systematic analogy that holds between a drawing and its meaning. To find the meaning *m* for a picture *p*, we must find a model of *R* which is isomorphic to translations of both *p* and *m*:

```
rg;p ≡ r ≡ rs;m
```

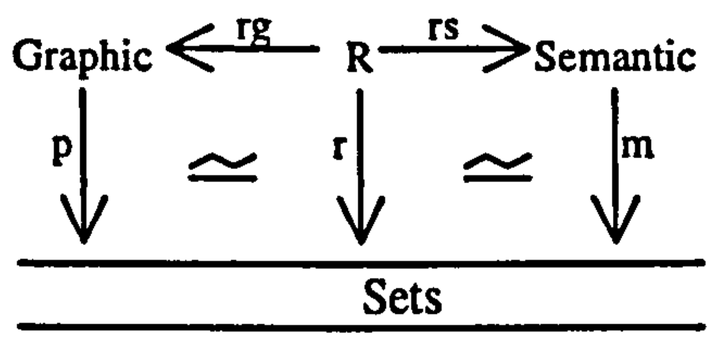


Figure 4.4 Displaying and interpreting

– as pictured in [fig 4.4]. The construction of the intermediate model *r* from *p* is a matter of deduction. However the discovery of a meaning *m* is not possible for all *r*. When it is, we may say that *p* is semantically well-formed. In general the translation *rs* yields a meaning that does not determine all aspects of the intended subject.

For a simple instance suppose that rectangles are a salient feature of drawings, and that they represent 'processes' which are known in the subject domain. Now given some drawing (in the

form of a description of the arrangement of its primitive elements, such as line-segments) we can ascertain the set of all rectangles that it contains. Then *R* will contain a sort *s* that *rg* maps onto sort 'rectangle' in the graphic theory, while *rs* maps *s* onto sort 'process' in the semantic theory.

Alternatively, for a pragmatic description, consider the task of issuing a token of a picture with its intended meaning in a suitable context. This is a matter of finding a context *k* that a picture *p* and its meaning *m* can be embedded into, via isomorphisms:-

$$gk;k \cong p \quad \text{and} \quad sk;k \cong m$$

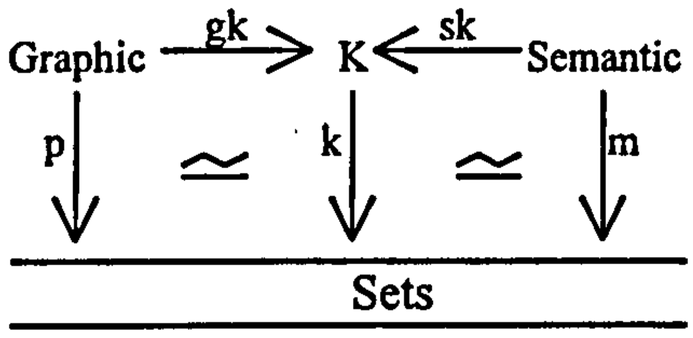


Figure 4.5 Expressions in context

– as shown in [fig 4.5]. The theory *K* determines those situations in which a picture may be presented and accepted. If the picture's meaning is to be controlled by theory *R*, then the isomorphism

$$rg;gk \cong rs;sk$$

(cf. [fig 4.1]) determines this constraint.

In this way a rudimentary framework for notation semiotics is provided via standard mathematics.

4.5.2.6 Semiotic Sorts and Graphoid Forms

Returning to the view of expressions as models of a theory, we may now focus on the shape of the graphoid forms as defined above (§4.4.4) . Since this work aims to develop notations that present theories about such forms, a similar argument to that of (§4.4.4) about the operations needed for building theories, suggests that the theories themselves should be categories whose objects are sorts. The forms are models belonging to some *topos*.

The kind of topos chosen to model graphoid forms is determined by the need to analyse expressions into a finite set of sorts of part. By our assumption of finiteness, we wish all expressions to be built from certain generating items. These items are maps from any of a finite set of simple forms – called syntactic, or more generally, semiotic *sorts*. If *J* is a sort and *A* is a form, maps $J \rightarrow A$ are called J-*items* of *A*. Thus any form is a finite configuration of such items.

> Generally, maps between forms are called terms; any term $f: A \rightarrow B$ is said to be *generated* if it is determined by a map from items of *A* to items of *B*. Terms are built up from *constructor* functions that

are given in the extra structure of the category. Thus a binary product allows us to construct ordered pairs `(p,q)` of items, and thereby define terms for binary relations and operations.

Noting that each sort corresponds to a set of items in a form, and that maps between sorts correspond to functions between sets of items, we see that graphoid forms are *functors* from *T* to a category of finite sets and functions. These considerations lead to the proposal that the forms be modelled as the topoi that are functor categories from theories.

A graphoid form is in fact a model of a theory *T* within a simple doctrine. *T* will be some category with a finite set of objects (sorts), and whose arrows are generated from a finite set by composition. The doctrine admits any functor from *T* to *Set* as an acceptable model: a *T*-graphoid form. These forms cover a wide range of notions of structure, and provide an uniform basis for definition of notations within a variety of logical doctrines, according to the complexity of the syntax. Provided that a syntactic theory is generated from some finite category *N*, every *N*-expression has such an underlying form, because every model determines a functor from *N* to *Set* – an *N*-graphoid. In other words we regard expressions as forms constrained by further properties – given as equalities forced to hold between constructed arrows – where the construction is permitted by the doctrine. This will be illustrated in the next chapter.

4.5.3 Coda

These final comments look back on the achievements of the chapter, so that we can reflect on their importance and the strength of the arguments made.

4.5.3.1 Summary

The work of this chapter has set limits on the area of study for a thesis concerned with applying mathematics to semiotic problems. Graphical notations of interest here are defined as sign-systems whose expressions can be drawn or written and which belong to a technical culture of software development professionals. To formalize is to explicate the rules that define notations – which, in notation design, makes possible the removal of ambiguity.

An exploration of the subject of semiotics has drawn attention to the structure of codes and the processes that motivate and maintain them. By adopting a semiotic perspective, it is found that expressions encode information by means of a tenuous logical association between graphical structure and meaning. This association is characterized by layering phenomena, described in

terms of articulation and connotation. These phenomena are governed by limits on logical complexity due to general cognition and the skills common to the cultural group that favours the notation.

This chapter has represented the form of expressions by means of a *generalized* type of graph or relational structure, sufficient to support rewriting and computation on parts of a whole. This graphoid structure is not so much a natural property of expressions as a consequence of the way we seek to operate on them. It should therefore be adequate in scope for describing all notations of interest – those that represent finite configurations, rather than those with continuous variation such as geographical maps, which would require topological operations. This is not, however, an absolute or dogma – if expressions were to be analysed from some other viewpoint, a different or more effective notion of structure could well emerge.

4.5.3.2 Grammar as Proof Strategy

As the approaches of computational linguistics suggest, both grammars and logical perspectives are encompassed by defining syntax as a theory in a logical language, and then introducing rewrite-rule schemes and constraint resolution as *proof strategies.* Parsing is a way of proving that a given graphical form is well-formed, by constructing from it a complete syntagmatic form. Interpretation proves the existence of a semantic form consistent with the syntagmatic form. By treating expressions as graphoid forms we fix on a type-theory within which to implement the logic and carry out these process efficiently.

4.5.3.3 Thinking with Diagrams

In (§4.5.1) it was suggested that diagrams, as much as formal narratives and formulae, are logical expressions that enable reasoning. The theoretical development here has concentrated on the logic of semiosis, on which this use of notation relies. In so doing it has postulated a continuum of inference through from graphical phenomena to semantic concepts, which goes some way to explaining how learned rules of manipulation can coexist with analogical graphical mechanisms, both helping the thinker to make their intuitions concretely verifiable.

This premiss has made it possible to put forward a plan for formalization that treats the expressions in a given notation as logical models defined by a formal theory.

4.5.3.4 Tectonics

A brief outline has indicated how a full notational theory may be built, around a theme of semiosis as a system of logical doctrines, theories, models and implementations. The proposed 'tectonic' theory offers a foundation for describing notation systems in context, in order to support operations of generation, translation and interpretation of expressions. Structure is definable by a logical presentation of a syntactic *theory* whose *models* are expressions (or situated expressions, in the pragmatic case). The generation of expressions is seen as the building of a model, guided by the theory, but directed towards semantic goals of the producer. Category theory has supplied a mathematical basis.

These ideas are stated in the widest generality. The following chapters are concerned with filling in some detail by studying specific cases and problems. They will show how to present the syntactic theories themselves as schematic expressions, and how to build theories for the purpose of designing or adapting notations.

Chapter 5

A Strategy and Notation for *Sketching* Syntax

Abstract

Here we find described a practical method for specifying syntax, which is based on the theory proposed in the previous chapter and can be uniformly applied to any notation. The strategy offers a basic technique of description that is intended to support the tasks of designing and processing notation.

The method applies the mathematical notion of *sketches*, which are formal presentations of theories. *Doctrines* classify the power of theories, by determining which logical constructions and inferences are admitted. Varieties of *category* are then represented as sketches closed under logical operations; a formal theory is defined as a category: the closure of any sketch that presents it. A schematic notation is then devised for drawing the sketch of a syntactic theory. Schemas are introduced to explain and depict all constructions on sketches, and to show how the presentation of a theory is built up. After some simple examples of notational structure, schemas are given for the syntax of Jackson Structure Diagrams.

This demonstrated method provides a formal graphical notation for syntax specification, which is equivalent in power to first-order logic. More importantly, it gives a way of controlling the complexity of the logical relations between segments and layers in syntax, and of defining direct structural analogy. It improves upon graph grammar approaches by making explicit the logical properties that remain hidden in the rules of a grammar.

The formalism encourages a view of syntax in which perceived connectivity constraints in an expression can be separated from the properties of a pictorial realization. The schematic notation is proposed as a research tool for analysis of notation, and is one of many possible. For practical purposes, some ways are suggested for extending and varying the notation according to circumstance. Alternative choices are considered.

Chapter 5.

A Strategy and Notation for *Sketching* Syntax

> *"The more that is left unsaid, the more possibilities are allowed by what is said. To determine what is entailed by what is said, all of these possibilities have to be covered one way or another."* (Levesque 1988, quoted in Stenning & Oberlander 1992)

The investigations of the preceding chapter have shown that the structures of graphical syntax can be founded on established mathematics rather than on *ad hoc* technique or theory. We wish now to show how the suggested mathematical methods can be applied to produce clear definitions for syntax – of both textual and diagrammed notation – by depicting the precise logical connexions linking appearance, syntax and semantics. We saw in (§3.2.1) that other proposed formalisms for specifying syntactic constraints on diagrammed notations mostly use textual notations. In preference, this chapter demonstrates a *graphical* approach to formally describing notations, by means of the theory of ***Sketches***.

The specific technical notion of a ***sketch*** is introduced and its theoretical background in Category Theory is explained. Sketch Theory is elaborated to make a formal framework and method for treating any semiotic layer as a set of constraints on a structured form. The framework naturally gives rise to a schematic notation – called SIGN – for formally specifying notational syntax. Some simple examples illustrate the concepts and symbols of the notation SIGN. Further examples show the strategy employed in depicting syntax, leading to a detailed study in which a syntactic sketch is given for Jackson Structure Diagrams. The advantages and problems in the design of SIGN are discussed in the light of the experience gained in the example specification task.

Why devise a graphical notation for specifying graphical notations? We have noted in (§4.3.1) that researches indicate an advantage in using diagramming to aid reasoning, provided the coding relies mainly on direct analogy. Suitable diagrams should thus be able to offer help in thinking about the structure of a notation being specified. Can we expect this reasoning to be within reach of an user's understanding? The arguments of (§4.4) show that the logic needed to define syntax is inherently tractable. Therefore there is good reason to believe that a graphical notation for syntax is appropriate.

It does not follow, however, that description of syntax is an easy task. When an user is *engaged* in working with a notation, in the sense of (§4.1.3), syntax is something that would properly rest

below their threshold of awareness. We should not expect it to be a natural task to focus on syntax and make its details explicit.

5.1 Theoretical Background

This section introduces the ideas behind the proposed *formal graphical* notation for syntax, based upon the mathematical Theory of Sketches as originated by Ehresmann (Bastiani & Ehresmann 1973) and developed in (Barr & Wells 1985, Gray 1989b). The concept of a sketch is also used below as a way to explain the structure of various kinds of category, which are viewed as closed forms of sketch. Whilst some categories are seen to be formal theories, others provide a home for models.

5.1.1 Sketching the Syntax of Notations

Throughout this chapter we have in mind a single 'syntactic' layer mediating between graphical and semantic aspects. As proposed in the preceding chapter (§4.4.1), the chosen approach seeks a description that specifies expressions as well-formed forms (wff) in their syntax, by stating constraints which restrict some class of combinatoric forms. Constraints are properties specified to hold on parts of any form that is an expression; they provide a *decision procedure* for well-formedness of a generated form: a means of checking whether it satisfies the required properties.

5.1.1.1 Syntax as Theory, Presented by *Sketches*

As suggested at the end of the previous chapter (§4.5.2), specific syntax can be regarded as a *theory* which names important sorts of syntactic *item*, and defines how these are related. The immediate difficulty here is that a theory is an ideal notion that allows for an infinite range of items and relations. To restrict ourselves to practical cases, we must presume that there is some finite set of primitive sorts of item and relation, from which the whole theory can be generated by procedures of logical deduction. This requires that we choose a finite *presentation* of the theory – which is the purpose of a *sketch*. Expressions are then definable as the models of a certain syntactic sketch.

What is a sketch? Sketches began as a method of presenting and studying mathematical theories; they are fundamental and general in purpose, with pictorial possibilities that suit them to the task at hand. They have already been investigated in computer science as a way to specify

algebraic data types (Wells & Barr 1987, Gray 1989a). Being founded on the logic of Category Theory, they are mathematical objects which can be transformed, combined and reasoned about. *Sketches* are in effect 'drawings' of theoretical structure. They can be expressed pictorially since their basic form is similar to the familiar *entity-relation approach to system description – but with* all relations constrained to be *maps* for *total functions.* The network of entities and maps defines a combinatoric form which is presumed to underly all the expressions. Sketches also apply constraints to this network.

Given a diagram or expression, a graphical *item* is a recognized configuration of marks and attributes located within it. Such items are perceived to be combined by various spatial and pictorial relationships. In *sketching* a syntactic theory, sorts of item are presented as *entities*, and the relations are described by *maps*, with each map standing for a specific kind of relationship. In this approach, however, the actual pictorial formations which realize these entities and maps are not analysed. Syntactic structure is based upon *combinatoric* structure or *connectivity* apprehended by a viewer, rather than upon geometry – we are not be concerned with intrinsic or analogical structure at this point. Cognitive habits of visual perception and interpretation pick out further 'natural' patterns of connectivity (emergent structure), which are also captured as entities in a *sketch.* Because of this visualization, the theoretical concepts involved in *sketches* are easier to grasp than one might expect for such a general and abstract approach, entailed by a rather austere[1] logical system.

5.1.1.2 Doctrinal Meta-Theory

Sketches are of different kinds, known as *doctrines*, which can be placed in a complexity hierarchy. The kind most commonly applied in computer science is the doctrine of *Finite Limit* (FL)-sketches; these define theories that are called *essentially algebraic.* In an algebraic approach, we might use an FL-sketch to specify a whole class of structures as a single model – often using the *initial* model that is generated by iterating syntactic constructor operations. Contrary to the intention here, an algebraic technique deliberately hides the internal structure of the algebraic elements. Further, the well-formed classes that we encounter are not usually easy to

---

[1]The logic uses a very small number of operators, though a richer set could be made available at the cost of extra notation and rules.

construct in such a manner.

In order to describe the internal structure of expressions, here will be adopted a broader doctrine whose sketches are called *mixed* or *general* in the literature. Within mixed sketches, the FL-structures are still important if we want to specify how part of a form can be generated by a series of operations.

The method of sketches enjoys a particularly strong and flexible kind of formalization. Not only does it provide a way to define syntax as a formal theory, but it also furnishes a formal meta-theory which allows us to view the syntactic theory either as a 'Platonic' object or as a formal construct in the meta-theory. As indicated below, and in (§6.1.1), a doctrine may itself be defined by an FL-sketch whose models are categories of the required type for the class of sketches. This two-tiered formalization enables reasoning about expressions, notations, and syntactic complexity. The doctrine may be presented in many different ways, with each particular FL-presentation resulting in a specific structuring of sketches. It is this flexibility that allows us the freedom to design a schematic notation.

5.1.2 Sketches

The following short exposition of the notion of Sketches, as it is applied in this thesis, is adapted from full accounts which can be found in (Gray 1989b, Wells 1987, 1994). (See also Coppey & Lair 1984, 1988; Makkai & Paré 1990). For the rest of this section, technical terminology is printed in bold italic (also see footnotes), while the corresponding terms used later in this chapter are underlined.

5.1.2.1 The Anatomy of *Sketches*

A ***sketch*** is a symbolic structure intended to be interpreted in a founding category **Z**, the latter serving as a conceptual modelling medium or paradigm. Unless stated otherwise, **Z** will here refer to some category of finite sets and total functions, that is generated by items of many kinds, and all ways of associating one item with another. This is what is properly meant by saying that an entity denotes a set, and a map a *total function* between two sets that are its ***domain*** and ***codomain***.

5.1.2.2 Signature and Constraints

A sketch in any doctrine can be analysed as a pair, `<signature, constraints>`. The *signature* is a directed graph *G* of entities and maps that determines the underlying form of expressions. These entities and maps specify sets and relevant functions – or, in a computational view, sorts and operations. It is best to think of a map as a dependent variable over elements of its domain, or as a '***local*** element' of its codomain – local to its domain. The usual elements of a set (which are called ***global***, i.e. constant) are represented by maps from some singleton set (Barr & Wells 1985). Each entity is both domain and codomain for an unique ***identity*** map, denoting the identity function on the set, which maps each element to itself.

A constraint is a kind of proposition that "mentions" parts of *G*. It is specified by a ***diagram-shape*** – displaying a ***graph morphism*** into *G* – simply a graph whose entities and maps are copies of those in *G*, connected in a way that respects the incidence properties of *G*. Doctrines differ in the kinds of constraint that are admissible.

5.1.2.3 The Doctrine of Mixed Sketches

The most general sketches considered here belong to the *mixed* doctrine, which supports three kinds of constraint. The first kind presents equality of functions denoted by paths of maps mentioned in the diagram-shape. The second constrains certain maps *from* a chosen entity (a part), and the third constrains certain maps *to* an entity (a piece). These formal constraints, usually known as ***diagrams, cones*** and ***cocones***, are each described below; simple examples of all the constraints will be encountered in the succeeding section.

Equalities

A *path* in a diagram-shape *D* is a sequence of maps; each path denotes the composition of the functions denoted by its maps. An equality on *D* (a *diagram*) expresses the constraint that the diagram ***commutes***:

> A diagram *D commutes* when any parallel paths (i.e. having same start and same finish in *D*) denote equal functions.

Without loss of generality we need only consider equalities between two paths.

Parts and Pieces

The dual notions of *cones* and *cocones* specify constructed sorts, denoting patterns within the expression. They respectively allow regular parts and whole pieces of an expression to be defined and their occurrence restricted.

Cones are diagram-shapes in which one entity – the apex – is connected by single maps – the sides – *from* the rest of the graph – the base. (This shape is perhaps more reminiscent of a pyramid on a shaped base). The apex represents a set of parts defined by the base in a natural manner; each side denotes an unique projection function from the apex set to a base set. In addition, each triangular face of the cone, formed from two sides and a base map, will be an equality diagram of the sketch.

The apex of a cone denotes a set of instances of some simple fixed combination; in a sense the 'largest / least redundant' or ***limiting*** set which satisfies the face equalities. Items of this set are a kind of part, whose fixed shape is laid out in the base.

Cocones are similar diagram-shapes in which the sides connect the apex *to* the rest of the graph – the base. The apex represents a set of pieces defined by the base, with projection functions from the base set to each apex set. Each triangular face of the cocone will be an equality diagram of the sketch. The apex of a cocone denotes a set of separate components, in a sense the 'smallest / most detailed' or ***colimiting*** set which satisfies the face equalities. Items of this set are maximal connected pieces, which may be of varying size.

Without loss of generality we need only consider certain simple bases for cones and cocones.

5.1.2.4 A Hierarchy of Doctrines

The nature of a doctrine is determined by which kinds of logical constraint are admitted in its sketches and which kinds of category are able to contain models of these sketches. Stronger doctrines admit more kinds of constraint.

(Wells 1994) lists various kinds of sketch that have been found useful, all of which are weaker than mixed sketches. The simplest of these are *trivial* sketches, consisting solely of a signature graph. In this thesis the models of signatures are the structures that are generically termed graphoids. The remaining kinds are used to specify various algebraic structures. A *linear* or elementary sketch may have equality diagrams; its models are algebraic structures whose operations are all unary (sometimes called ***pre-sheaves***). A *finite product* sketch has cones over finite discrete diagrams, with models that are multisorted universal algebras given by finite signatures and equations. A *finite limit* sketch has finite cones, but no cocones; this kind corresponds to (Freyd 1972) ***essentially algebraic*** structures, and includes all Horn theories (Barr 1989). A *finite sum* sketch has finite cones and finite discrete cocones. Where necessary these kinds will be abbreviated to L-, FP-, FL- and FS-sketches.

Formally, a ***doctrine E*** corresponds to a type of category definable *essentially algebraically* over the category of categories [Lawvere]. Categories of this type may be called E-categories, and structure-preserving maps between them are called *E-functors*. An *E-sketch* then allows the specification of any

kind of construction that can be made in an E-category – if an E-sketch admits a certain kind of (co)cone, an E-category must have the corresponding kind of (co)limit. This is the same as saying that a doctrine may itself be sketched by an FL-theory, which opens up many avenues for generalizing the concept of sketch beyond Ehresmann's definition – see (Wells 1990) and further references in (Wells 1994).

5.1.2.5 Sketches as a System of Logic

Sketches can be placed in a more general setting, within Goguen and Burstall's formalization of general logical systems as ***institutions*** – as described in detail by (Barr & Wells 1990). Briefly, in an institution, a theory is presented by a signature and a set of sentences in some language – in our case a sketch has a diagrammatic set of sentences. A relation of satisfaction holds between a model and a presentation if all the sentences are true for the model (Rydeheard & Burstall 1988; Goguen & Burstall 1984, 1986, 1992).

In the previous chapter a need to control the complexity of logic was emphasized (§4.4.2, 4.4.3). The finite mixed sketches (FM-sketches) that will be used here to specify syntax are more powerful than the algebraic systems (FL or FS) just described; Mixed Sketches are assessed as equivalent in expressive power to first-order logic (Makkai & Paré 1990)[2]. The necessary explanation of the categorial analysis of logic may be found in (Makkai & Reyes 1977, Pitts 1989). The sketches should therefore be powerful enough for any level of syntax, but too powerful for most purposes except semantics.

5.1.3 Categories

In (§4.4) and (§4.5), references were made to the theory of *categories*. Taking an unusual but instructive viewpoint, categories are here treated as sketches *closed* under the rules of deduction specified in some doctrine. Instead of regarding a category as a structure with observable properties, we view it as a full collection of formal objects that express these properties.[3] This will allow the discussion of sketches, theories and larger categories in a common framework. For a thorough treatment of Category Theory the reader is referred to (MacLane 1971).

---

[2]However Wells (1994) notes that formal coequalizers are a little stronger than FOL: the category of connected graphs cannot be specified in finite FOL formulas and terms.

[3]Thus a category is confused with its image under the underlying functor **Cat** → **Sk** from categories to sketches.

5.1.3.1 Categories as Completed Sketches

A *category* is an algebraic structure, defined as indicated in the previous chapter (§4.4.4). The standard definition corresponds to an L-category **C** which is a *sketch*, with only signature and equalities, that supports construction of composed maps. It satisfies two closure properties:-

(1) Closure under composition of maps:

If maps f: A → B and g: B → C exist in **C** between the entities A, B, C, then there exists a map h: A → C and an equality h = f ; g .

All the maps which can be constructed by composing existing maps, must be included in **C**.

(2) Closure under inferred equalities.

All equalities which can be inferred within **C** must be included in **C**. These equalities follow from the associative property of composition and the existence of an identity map for each entity (see §4.4.4). Entities in a category are called ***objects***, and maps are called ***arrows*** or *morphisms*. Normally the complete sketch would be called the *underlying* sketch of the category.

Stronger closure properties are needed to define M-categories, where *M* is the doctrine of mixed sketches. In any category, a *(co)limit* is an object constructed as the apex of a (co)cone. A category is ***complete*** if it satisfies:-

(3) Closure under limits – every base graph definable on the signature must be the base of some cone in **C**.

Similarly, a category is ***cocomplete*** if it satisfies:-

(4) Closure under colimits.

M-Categories are those that are ***bicomplete*** – both complete and co-complete.

The apex of a (co)cone is often called 'the' limit of its base; although there may be many limits on the same base, they are all isomorphic. Sometimes it is convenient to assume that in a category there is always a favoured apex that can be called the "canonical" (co)limit of the base, though this is not strictly in keeping with categorical methods.

5.1.3.2 *Large* and *Small* Categories

The particularly important category **Set** has all sets as entities, and all total functions as maps. In mathematics **Set** is used as a foundational 'semantic universe' in the sense that it is usual to explain all concepts in terms of sets and functions. Such categories are technically described as

large, which is to say that there is no consistent way to define the class of its objects as a set – owing to Russel's Paradox, the class of all sets cannot itself be a set. In usual terminology, even categories with an uncountable number of objects are called *small*.

5.1.3.3 Categories of Sketches and Categories

We can define a morphism between sketches as a map that sends entity onto entity and map onto map while preserving all formal constraints. It can be shown (Gray 1989b) that sketches then form a category **Sk** which is bicomplete, and also closed under further useful constructions. There is also a category **Cat** (seen as a subcategory of **Sk**) whose objects are small categories, and whose morphisms are called ***functors***.

5.1.3.4 Categories of Models in a Medium

A ***model*** of a sketch *S* is a sketch-morphism from *S* to some category whose closure properties support the doctrine for *S*. The models of *S* form a category which will be described in the succeeding chapter (§6.1.2). We shall especially consider models in a category **Z**, possibly large, which will have the role of a Modelling Medium – an abstraction for all the possible situations considered and in any context we might be interested in. For the formal purposes of our application, we need not resort to large categories, instead preferring a countable (recursively generated) category for **Z**.

In any case, because the medium is a category, it represents a mathematical idealization – an infinite completion. If we assume that we can obtain total information about any situation, we can take **Z** to be a suitably closed *full* subcategory of **Set** (i.e. some class of constructed sets with all its functions). Although this is the course taken in this thesis, it is a particular advantage of sketches that they apply equally to other philosophies, such as those that can cope with incomplete information (e.g. the category of topological spaces, or of computable sets).

We must be careful to distinguish between the ***internal*** formal logic defined by the doctrine and the *external* logic – 'accidental' properties and further logical regularities – that **Z** might also obey outside of the doctrine. **Set**, for instance, obeys stronger doctrines than Mixed Sketches, and would justify more constructions and inferences than are formally permitted in M-categories.

5.1.3.5 Theory Categories

Any sketch can be completed (by construction and inference) to form a *small* category known as its ***theory***. The theories nevertheless have a countable infinity of objects, that might loosely be thought of as concepts, or types of pattern. *Primitive* concepts are those entities/maps given in the sketch; *derived* concepts are those that can be constructed or inferred.

An E-sketch *S* generates a formal theory E<S> that is an E-category, in such a way that the sketch *S* has the 'same' models as its theory, and we can work with sketches rather than with the theories described at the end of the previous chapter. This procedure will be treated in more detail in the next chapter (§6.1.1).

5.2 SIGN: A Schematic Syntax Notation

Now that the theoretical groundwork is in place, we need to see how the foregoing ideas may be applied. The task of this section is to illustrate how sketches can serve as a method of syntactic description. For this purpose, a graphical notation (SIGN) is proposed, which draws mixed sketches as *schemas*. The proposal gives details of the method, using elementary examples to show how SIGN depicts sketch constituents.

The examples which follow show how the schematic syntax notation (SIGN) is used to describe structure which commonly occurs in diagrams; these are chosen to introduce *sketch*-concepts and the range of SIGN symbols. Some standard terminology from Set Theory and Graph Theory are used to explain the concepts; the more general but less familiar terms of Category Theory are mostly relegated to footnotes.

Many graphical notations contain, amongst others, structures recognizable as directed graphs. This notion of graph is a convenient starting point for explaining how to describe syntax. Following this, we are introduced to some common constructions which can be made on graphs. It turns out that directed graphs can serve as elementary units from which all the more complex structure of notations can be built[4]; taking advantage of this fact helps keep the schema notation simple.

---

[4]This is the consequence of some simple theorems in Category Theory: Any category that has finite (co-)products and all (co-)equalizers, has all finite (co-)limits.

5.2.1 Introducing SIGN

The schematic notation is proposed as a means of drawing the signatures, constructions and constraints of a syntactic sketch. As just explained (§5.1.1), the sketch *presents* a theory that defines syntax; forms are well-formed precisely when thay *satisfy* this theory. Here we take a first look at the basic structure of schemas, and at some sketches they can depict.

5.2.1.1 Formalized Entity-Relation Schemas

Following the above approach (§5.1.1), SIGN directly depicts entities and maps. It can be viewed as a kind of Entity-Relation notation in which relations (maps) are pictured with the usual "crow's foot" (many-to-1) connectors, between rounded boxes that depict entities. As in other Entity-Relation diagrams, extra markings and links on the connectors serve to signify constraints on the relations. For an expression of the described notation, an entity in a syntactic sketch corresponds to a recognizable sort of situation or pattern element, and a map usually amounts to a visual or mental tracking operation on an expression, such as following a line, or associating an element as part of some perceptual or linguistic gestalt. An equality between maps describes the case where two tracking operations always have the same outcome.

5.2.1.2 A Simple Mixed Doctrine

Schemas are based on a particular doctrine of *mixed* sketches. The doctrine allows for a rich variety of constraints, but for practical reasons only certain basic cases are incorporated in the schema notation. The effect of this is to reduce the expressive flexibility, but without any loss of logical power. For instance, the only *commuting diagrams* used in schemas consist of a parallel pair of paths. One important case of parallel paths is an equality triangle, which can be used to construct a new map equal to the composition of two successive maps.

The parts used in schemas are based on just five types of cone and corresponding cocone, defined on simple base shapes and named as follows:

'Parts' used in schemas:-	singleton, product, loop, pullback, injection,
Cone types:	***terminal, product, equalizer, pullback, monic.***
'Pieces' used in schemas:-	zero, disjoint union, component, pushout, surjection,
Cocone types:	***initial, coproduct, coequalizer, pushout, epic.***

The basic cases are all illustrated below (§5.2.2).

5.2.1.3 Directed Graphs

Our first example of a sketch defines diagrams that have a directed graph structure, requiring only a single *schema*.

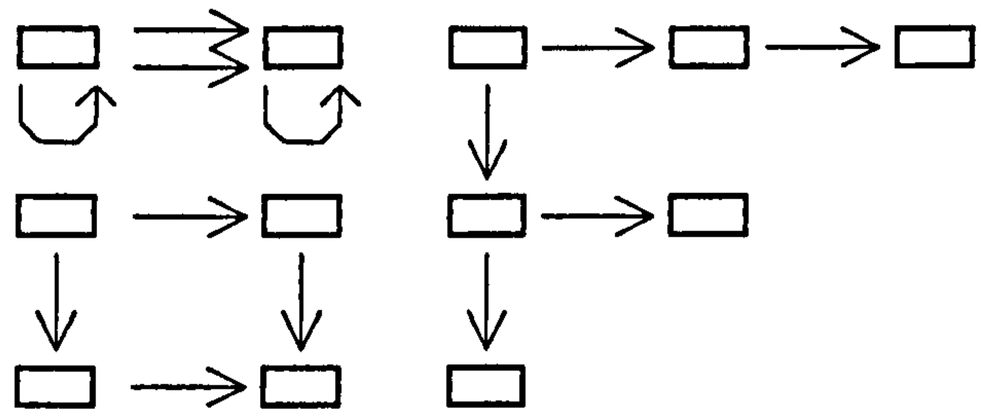


Figure 5.1 An example Directed Graph diagram

Consider a diagram (for example [fig 5.1]) which presents a connective structure known as a directed (multi-) graph or "digraph". It is made out of two kinds of pictorial item: rectangles and arrows. The arrows are directed links, and they may be drawn without restriction between any pair of rectangles. It is assumed that the only significant information in the layout is the connective effects of the arrows.

Each entity in a syntactic sketch generally corresponds to a set of pictorial items on a diagram; these items are tokens: often basic shapes or pattern elements that are seen in the diagram:-

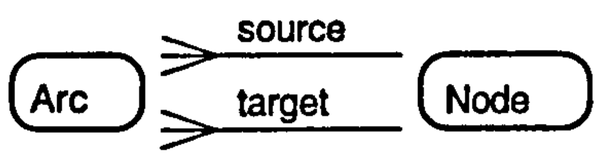


Figure 5.2 A Sketch for digraph syntax

The sketch, depicted as a single schema [fig 5.2], shows sets of rectangles and arrows as the two entities labelled *Node* and *Arc*. These entities are visible in the sense that they are actual sets of marks on the diagram [fig 5.1], rather than abstract properties or hidden information known to the reader.

We can refer to items on the diagram either by their syntactic roles (*nodes* and *arcs*), or equivalently by their depictions as rectangles and arrows. More generally, syntactic names are best regarded as describing situations which can be recognised from graphical configurations. In an actual notation, we could also refer to items by their semantic values, i.e. what they stand for. (E.g. a node might represents a system-unit, and an arc a channel that transmits control signals.) Naming thus reflects the layer of syntax that we have in mind.

Each map linking entities in a sketch signifies a total function between sets, *implicit in the diagram* as a recognizable incidence relationship or association between items:-

In the diagram [fig 5.1], each arrow is associated with the rectangle adjacent to its tail (simply by virtue of proximity); this property is shown in the sketch [fig 5.2] as a map called *source* – every arrow must have an unique source rectangle at its tail. In the same way, the map *target* denotes the association of each arrow with the rectangle at its tip. Maps *source* and *target* state the two roles that nodes may take

in association with arcs.

The sketch [fig 5.2] is all that is necessary to capture the syntax for the whole class of digraph diagrams.[5] The two maps express connectivity.

Two maps in a sketch which (like *source* and *target* in [fig 5.2]) share the same domain and the same codomain are called parallel. It is this situation that characterizes the presence of a digraph structure.

5.2.1.4 Decomposition and Labelling

Because they express only connective properties of maps, sketches could be drawn entirely as unlabelled structures. In practice, large sketches easily become "cans of worms", when many connectors cross over each other. In order to avoid illegibility, ambiguity and error, it is more convenient to notate a sketch by *decomposing* it: drawing it as a set of separate schemas with labelled boxes and connectors. [Fig 5.2] shows a simple schema. The box labels stand for entities and the connector labels stand for maps, of the resulting sketch.

Unlike (some uses of) entity relation diagrams however, labels carry no meaning, i.e. they hide no information, but are there only for ease of reference. Relabelling a set of schemas does not change the *sketch* they express, provided that the distinctness of labels is preserved. Verbal labelling carries the added advantage that constraints can be read off in natural language or formulae, if desired. An entity label is usually a noun which names a type of item; a map label is a noun which sometimes names a role for an item of its codomain.

Each schema normally concerns some particular fragment of a syntactic signature, which is effectively a graph *morphism* into that signature, as is made clear in the examples below.

5.2.2 Canonical Constructions

Graphs exhibit observable patterns which often have some significance in a given notation: for instance components, loops and node-pairs. In order to describe a pattern in a notation, a *construction* is made on the sketch: a new entity representing instances of the pattern is added, together with more maps and a defining constraint.

This next example shows how entities and maps can be constructed *canonically* (i.e. by "natural"

---

[5] In passing, notice that this sketch is itself in the form of a directed graph.

rule) from neighbouring maps [fig 5.3]. The same example diagram will be used [fig 5.1], but with an enriched interpretation, by drawing attention to a pattern that a viewer may easily recognize and give meaning to. Such constructions extend the syntax, even though the appearance of the notation does not change. Our first constructions are Components and Loops, which are shown to be *dual* to each other. The notion of an equality between paths is also illustrated.

5.2.2.1 Graph Components

To define the components of a graph, the sketch requires a certain *cocone*.

A component of a graph is defined as a maximal connected subgraph. It is easy to see the three components of [fig 5.1], and verify that every rectangle is part of some unique component. (This is the criterion for a *total function* on sets.)

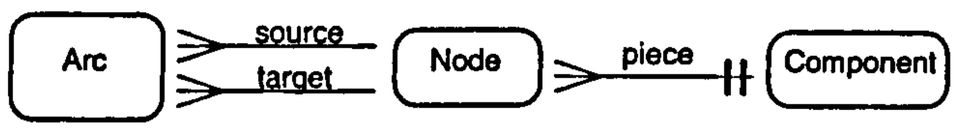


Figure 5.3 The Component construction

In the sketch [fig 5.3], the entity *Component* has been added. It also has a map *piece* from *Node* which associates every node of the graph to the component of which it is part.

The graphical realization of this map is a 'part-of' relationship.

In the extended sketch, the new entity and map are constructed by a rule of logic from the simple sketch [fig 5.2], using a further symbol to express this *canonical* relationship.[6]

The double bar at the head of the connector denoting the map *piece* signifies that it and *Component* are defined canonically in terms of the parallel maps *source* and *target*, which define the connectedness of the graph, and hence fully determine the map *piece*.[7]

5.2.2.2 Paths: Compound Maps and Equalities

A sketch specifies certain constraints as *equalities* of functions. The maps in a sketch frequently form sequences, known as *paths*, which are compound maps denoting *compositions* of functions. Just as there are parallel maps, there can be 'parallel' paths. The functions depicted by two parallel paths may be specified equal, and there is a special notation for this.[8]

---

[6]The Idea of construction is an analogy with Euclid's methods, for example in geometrically constructing a perpendicular line and marking the rightangle.

[7]The map *piece* is called the **coequalizer** of *source* and *target*, in Category Theory, because of the equality in fig 3.4.

[8]This pictorial notation for equations is a very powerful feature of *sketches*. It is a good visual aid to reasoning.

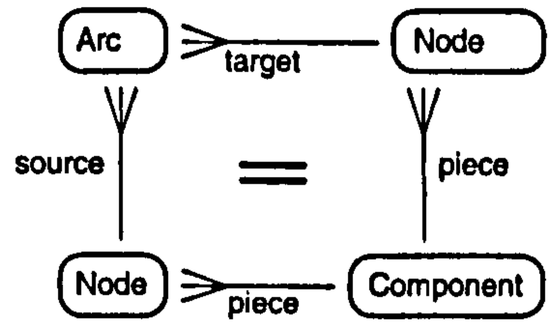


Figure 5.4 Two equal paths

Equal paths

Notice that each arc also belongs to a unique component, namely the component of its source node; this association is in the sketch as a compound map, *source* then *piece*. Equally, the same association can be found from the component of the arc's target node: the map resulting from the compound *target* then *piece*. This equality

`(source;piece = target;piece)` is a canonical property of the map *piece*: a logical consequence of the component construction. It can (if desired) be expressed in a further schema [fig 5.4], which shows the equality by enclosing an equal-sign between the two paths, separated by duplicating both the entity *Node* and map *piece*.

The cocone construction places *Component* on the apex and takes the entities and maps of [fig 5.2] as base. The triangle face equalities of the cocone result in the equality of [fig 5.4] – as detailed below in (§5.3.1), [fig 5.23].

5.2.2.3 Graph Loops

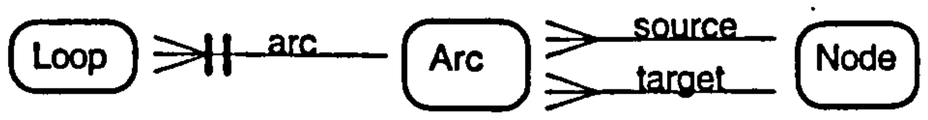


Figure 5.5 The Loop construction

To define the loops of a graph, the sketch requires a certain *cone*.

In [fig 5.1] there are two arrows that curve back from a rectangle to itself. These are both examples of a *loop*: syntactically, that special kind of arc whose source is also its target.

[Fig 5.5] extends the sketch [fig 5.2] with the entity *Loop* and a map *arc* which expresses the graphical inclusion of the set of looped arrows in the set of arrows (the subset relationship: every *loop* is an *arc*). The canonical map *arc* associates the loop (an instance of a property) with the arc that has this property[9]. The map *arc* is drawn in line with its defining parallel maps *source* and *target* and shown with a double bar at its foot in [fig 5.5]. The two compound maps `arc;source` and `arc;target` have the defining property that they equally attach a loop to its node; this equality:

`(arc;source = arc;target)` is shown in [fig 5.6].

[9]The map *arc* is called the **equalizer** of *source* and *target*.

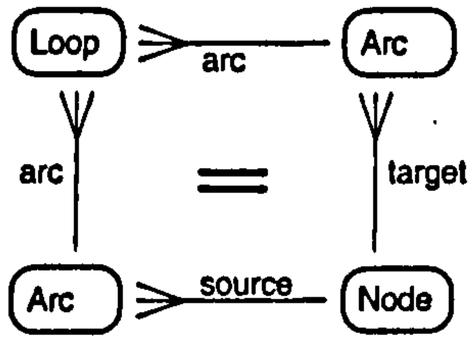


Figure 5.6 The loop equality

The double-bar notation for the loop construction is the same as for component [fig 5.3], except at the opposite end of the relevant connector. The reason for this is to emphasise the following symmetric aspect of sketch logic.

5.2.2.4 Duality

The last two constructions, loop and component are *dual* to each other, i.e. they are carried out identically except that the corresponding maps are reversed. This can be seen from [fig 5.7], which arranges schema [fig 5.3] above a re-ordered equivalent of [fig 5.5], in order to show the correspondence. Labels are omitted because it is the structure that is being compared, not the references. [fig 5.4] and [fig 5.6] also correspond in this manner.

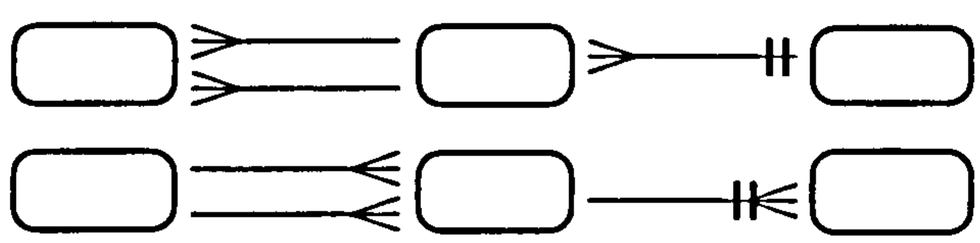


Figure 5.7 Duality of component and loop notions

Every sketch construction has a dual. Note that dual situations are by no means similar in appearance (e.g. loops do not look like components). This logical symmetry of sketch notation is a bonus because all deductive or constructive rules come in similar pairs, effectively halving the complexity of the formalism. The schematic notation preserves this symmetry in its choice of symbols; a dual situation is obtained by graphical reversal of connectors.

5.2.2.5 Cartesian Products

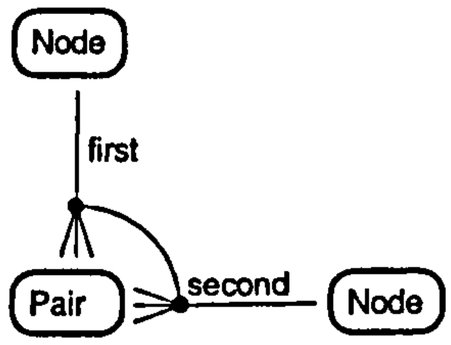


Figure 5.8 A Product

The last example of a canonical construction is the Cartesian product, (in this example) of a set with itself. It constructs an entity for the set of all ordered pairs, together with the projection maps

that extract the first and second members of each pair.

Node-pairs

Consider all pairs of rectangles in the diagram [fig 5.1]; a pair simply consists of any two rectangles, the first and the second (which may be the same). In the schema [fig 5.8], the entity *Pair* has projection maps *first* and *second* which express these assignments of each node-pair to its nodes. The pair construction is expressed by linking the two connectors denoting projection maps at their feet. In a diagram, these maps are realized as 'part-of' relationships, in which each node is part-of many pairs.

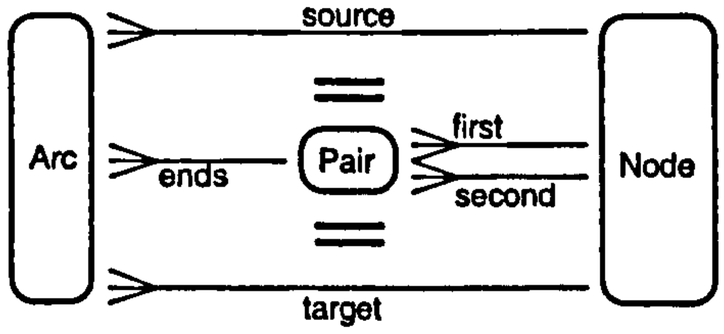


Figure 5.9 The pair at the arc's ends

Constructions are essentially about seeing a structure in a new way. This particular construction allows a digraph to be seen as a single map, from arcs to pairs of nodes.

Notice now that each arc is associated with the node-pair that it connects, i.e. that formed by its source and target; this is a canonical map *ends*, which is the unique map satisfying the two equalities expressed in [fig 5.9]: the *first* end is the source, the *second* end is the target:
`(ends;first = source)` and `(ends;second = target)`.

5.2.2.6 Syntactic Signature

Every notation covered by the theory of Chapter 4 is assumed to be based on a form which is termed a graphoid in this thesis. We can now see how to separate out the underlying graphoid structure of a syntax from its constraints.

In the example, we have arrived at a sketch that is depicted by four schemas [figs 5.3, 5.5, 5.8, 5.9]. What is the *form* of expressions according to the data in this example sketch? The maps and entities of the sketch formed by bringing together all the four schemas and identifying items with the same label, comprise the *signature* of the syntactic sketch. Without the defined constraints, this signature is a *trivial* sketch (§5.1.2), which specifies the graphoid form.

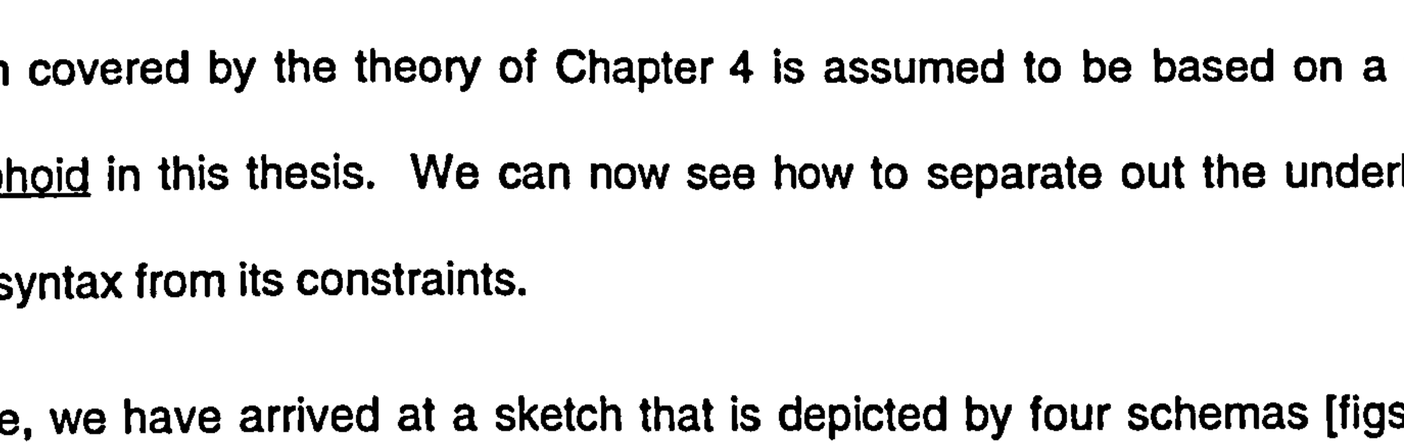


Figure 5.10 The Signature of the syntax

With the above constructions added, the syntax of diagrams like [fig 5.1] is based on the sketch [fig 5.10]. This shows all the entities and maps without constraint; it is called the signature of the digraph notation being described, and is an extension of [fig 5.2]. The four schemas [figs 5.3, 5.5, 5.8, 5.9] suffice to express the syntax, which logically consists of this signature together with all the depicted constraints.

The four schemas each refer to a fragment of the signature, defined by a graph morphism. A morphism is drawn as a graph (the domain of the morphism), with the mapping indicated by names of entities and maps in the codomain[10] – in this case the signature.

5.2.3 Further Constructions

Before leaving the digraph example, some further useful constructs are mentioned here, to give a full description of SIGN symbols. The first of these is dual to the Product, and gives yet another view of a digraph. The second allows one-to-one correspondence between items to be notated. Finally the construction for cardinal numbers zero and one completes the basis of sketch logic.

5.2.3.1 Disjoint Union

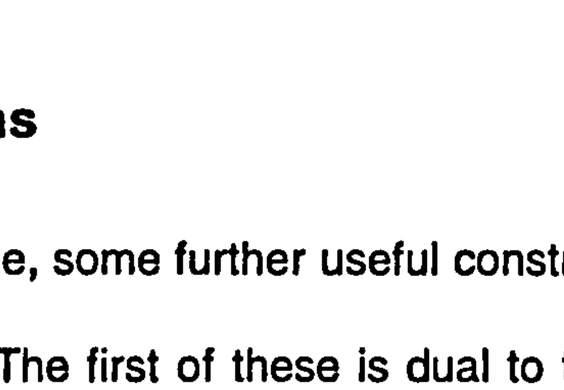


Figure 5.11 A disjoint union of *tip* and *tail*

Given two sets of items, we can form their union. Each set is a subset of this union, and is represented explicitly as an *inclusion* map: an injective (one-to-one) function that serves to mark a subset of its codomain (the union, in this case). The *disjoint* union of two sets is formed by taking disjoint copies of the two sets; the union of copies is codomain for two inclusion functions that represent these copies as its subsets. In the following example, the two sets are identical, making it clear that subsets are represented by maps in a sketch, not by entities.

Each arrow of a digraph has two extremities, referred to as *tip* and *tail*, which are found adjacent to rectangles. The disjoint union of *Arc* with itself is named *Tip&Tail* in [fig 5.11]; it denotes the set of extremities, either tips or tails, each associated to a particular node, its *place* in the diagram. The existence of the map *place*, which now carries all the connectivity information, is a logical consequence

[10] A schema is in fact a drawing of a *sketch*-morphism, because constraints on maps in the schema also correspond to those in the sketch.

of the construction, the unique map which satisfies the equalities:

`(tip;place = source)` and `(tail;place = target).` (cf. [fig 5.9])

The link symbol denotes Disjoint Union, as in its dual construct Product, but at the opposite end of the connectors[11].

This construct is useful for dividing a sort into partitions, or conversely, when separate sorts of item share common syntactic properties, to collects these into a recognized 'supersort'.

5.2.3.2 Identity Maps

Every entity has an *identity* map (meaning 'self'), which maps each item onto itself.[12] The name of the identity thus the same as its entity. When necessary, an identity map is drawn as a thick grey connector from the entity to itself, as in [fig 5.12].

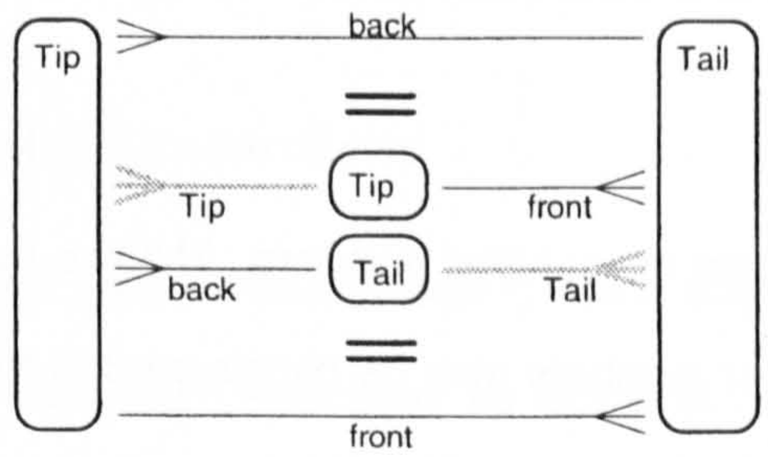


Figure 5.12 The bijection property

5.2.3.3 Bijection

The identity maps provide one way to define bijective correspondence between sets. Such bijections are found in diagrams whenever two graphical situations (pattern items) *always* occur together. The mathematical definition of bijective function translates into a pair of triangle equalities.

Tip and *Tail* denote the two disjoint sets which are both 'copies' of the set of arrows (*Arc*), in the sense that they occur in the exact same situations in which arrows occur.

The sketch [fig 5.12] represents tips and tails of arrows as two different entities. The bijection between them consists of two maps: *front*, which finds the tip joined to a tail; and *back*, which finds the tail joined to a tip.

---

[11]The dual of the **product** construction is normally called the **coproduct** in Category Theory. Here the term disjoint union is used to help the general reader's intuition – though the property of disjointness is not generally a formal consequence. Similar remarks apply to the use of 'injective' and 'surjective' where *monic* and *epic* would be preferable.

[12]In fact an entity can be thought of as a 'degenerate' map; maps ("arrows") are the primary concept on which Category Theory is based. The depicted signature of a sketch leaves all identity maps implicit.

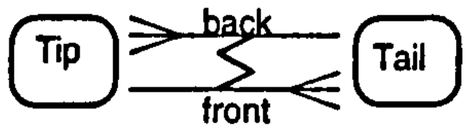


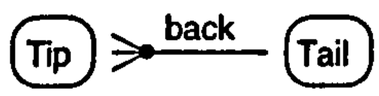
Figure 5.13 The bijection symbol

The bijection property is presented by two equalities:

`(back;front = Tip)` and `(front;back = Tail)`.

For convenience, a single symbol (a zigzag link) denotes this circumstance [fig 5.13].

Alternatively, a notion of isomorphism between sets denoted by *Tip* and *Tail* can be defined by constraining *back* to be the side of a cone whose base is the singleton graph having just the entity Tail – notated as a 'product' of Tail alone (with a spot above the foot of the connector):



This works equally if a co-cone is used instead (i.e. with the spot at the head).

5.2.3.4 Cardinal Numbers

Constructed entities can represent the cardinal numbers. The two basic cases, 0 and 1, are empty sets and singletons, while larger numbers may be constructed arithmetically, using disjoint union for addition, and products for multiplication. The numbers are thus constructed as abstract sets. Equality of two numbers is represented by any bijection between the abstract cardinal sets that 'depict' them.

The abstract entity **Zero** is constructed as apex of a cocone whose base is empty.

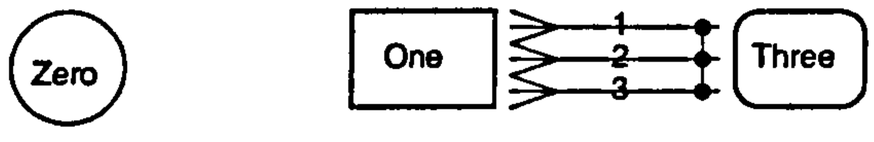


Figure 5.14 Cardinal Numbers

A set which is empty represents zero. The symbol for this is to use a circular boundary for the entity, a 'No Entry' sign that prohibits any item of this sort from occurring in the diagram.

The empty set *Zero* has a defining property:- To any entity *E*, there is a canonical map from *Zero*, and it is the only one. This map is essentially empty; it is conventionally named "*?E*".

The abstract entity **One** is constructed as apex of a cone whose base is empty. Thus *One* is dual to *Zero*.

A singleton set contains just one element. The symbol for this is to use a square-cornered boundary for the entity, having the effect of forcing there to be an unique item of this sort.

[Fig 5.14 right] shows the addition 1+1+1, a disjoint union construction of the number *Three*.

The singleton denoted by *One* has a defining property:- From any entity *E*, there is a canonical map to *One*, and it is the only one. This map sends each element of *E* to the only element of *One*; it is conventionally named "*!E*" .

Sometimes a syntactic rule restricts the size of a set. Cardinal numbers can be used for size comparison, with the help of an injective map (as defined below).

5.2.4 Some Derived Constructions

The symbols introduced above form a logical basis for structural description. As with other systems of logic, extra concepts can be built on this basis by methods of definition, as desired. To finish this section, four more constructs are defined which are frequently used and therefore warrant special symbols.

5.2.4.1 Pullback and Pushout Constructions

Included in SIGN are two useful (dual) constructions[13], which may be derived from those given above, in each case by viewing certain structures as digraphs. This derivation is given as a definition for the extra notation.

The *pullback* construction arises as the relation that holds between items of two sorts by virtue of items being arranged in clusters.

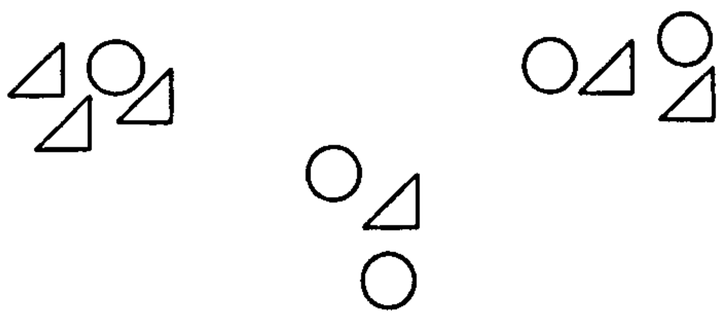


Figure 5.15 An example Cluster diagram

Figure 5.16 A Sketch for 'bi-clusters'

Bi-clusters

In a diagram like [fig 5.15], *Circles* and *Triangles* are jointly 'clustered' into *Groups*. This is sketched in [fig 5.16]. The first step of the construction is to form the product *Triangle* × *Circle*, and to view its pairs as the 'arcs' of a digraph whose nodes are the groups [fig 5.15]. In each pair, the triangle is the tail of the 'arc', and the circle is its tip; the arc's body is invisible. Accordingly [fig 5.17 left], the product entity is called *Arc*, and the parallel maps of the digraph are defined by the equalities:

```
(source = left;tri) and (target = right;circ).
```

---

[13] **Pullback** and **pushout** are used extensively in Category Theory .

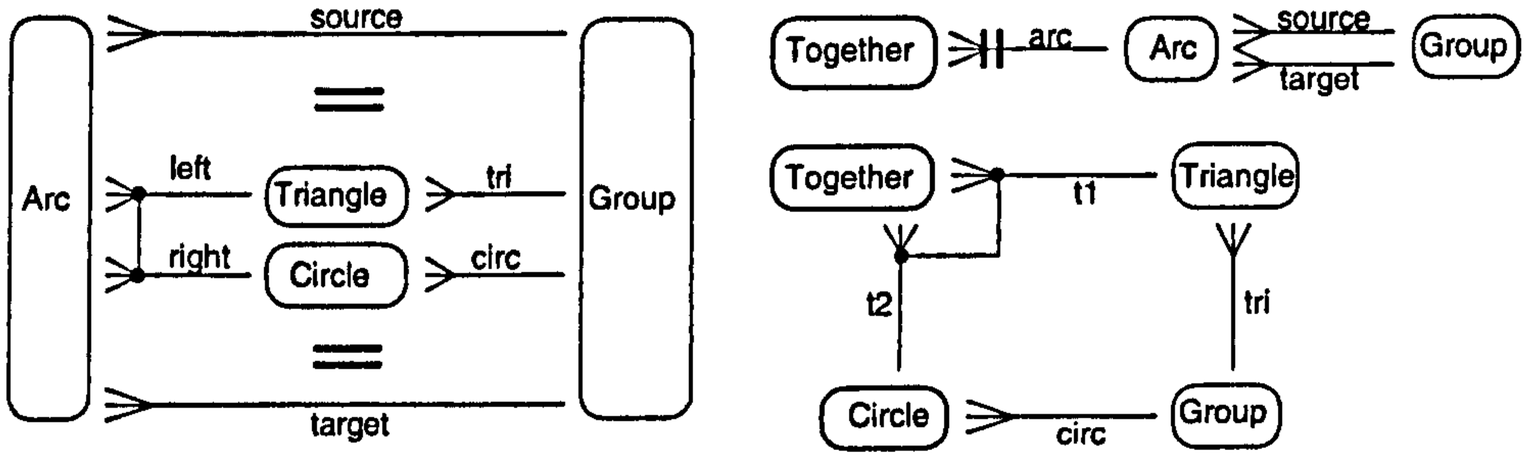


Figure 5.17 A directed bi-cluster graph. Constructing a togetherness relation

The next step forms a relation *Together* (*t1*, *t2*) that holds between any circle and square belonging to the same group; the relation is a 'loop' entity on the discovered graph, shown in [fig 5.17 upper right].

The maps are defined by the equalities (not shown):

```
(t1 = arc;left)  and (t2 = arc;right).
```

The chain of equalities shows formally that t1 and t2 are in the same group:-

```
t1;tri = arc;left;tri = arc;source
  = arc;target = arc;right;circ = t2;circ
```

The symbol for this is the 'rightangle' sign shown in [fig 5.17 lower right], linking the projection maps *t1* and *t2*. The map *t1* is termed the map obtained by 'pulling back' *circ* along *tri*, and *t2* is similarly the result of 'pulling back' *tri* along *circ*.

Dual to this is the *pushout* construction. This is based on components of a *bipartite* graph structure, which shows connexions between two sorts of items. The method is to view any bipartite graph as a directed graph, by ignoring the distinction between the sorts of node.

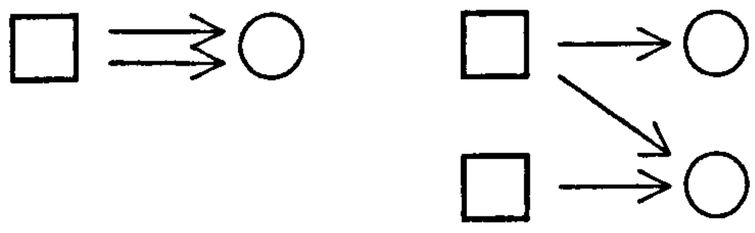


Figure 5.18 A Bipartite Graph diagram

Figure 5.19 A Sketch for Bipartite Graphs

Bipartite Graphs

Consider a diagram (for example [fig 5.18]) of a graph with two sorts of node, squares and circles, and arrows which can only run from a square to a circle. Here squares are drawn on the left, and circles on the right.

The sketch for a bipartite graph, which is dual to [fig 5.16], is depicted in [fig 5.19]. The required construction finds the components of the graph. This is done by forming the disjoint union of the two sorts (*Node* = *Square* + *Circle*), and constructing two new maps *source* and *target* [fig 5.20 left] defined by the equations:

```
source = left;s  and  target = right;c.
```

The components of this digraph are also the components of the bipartite graph. They can be described as in the construction above [fig 5.3].

The symbol used is the 'rightangle' sign shown in [fig 5.20 lower right] linking the two canonical

projection maps f and g. The map f is said to be obtained by 'pushing out' *right* along *left*, while map g is obtained by pushing out *left* along *right*. This entails the equality: `right;g = left;f`.

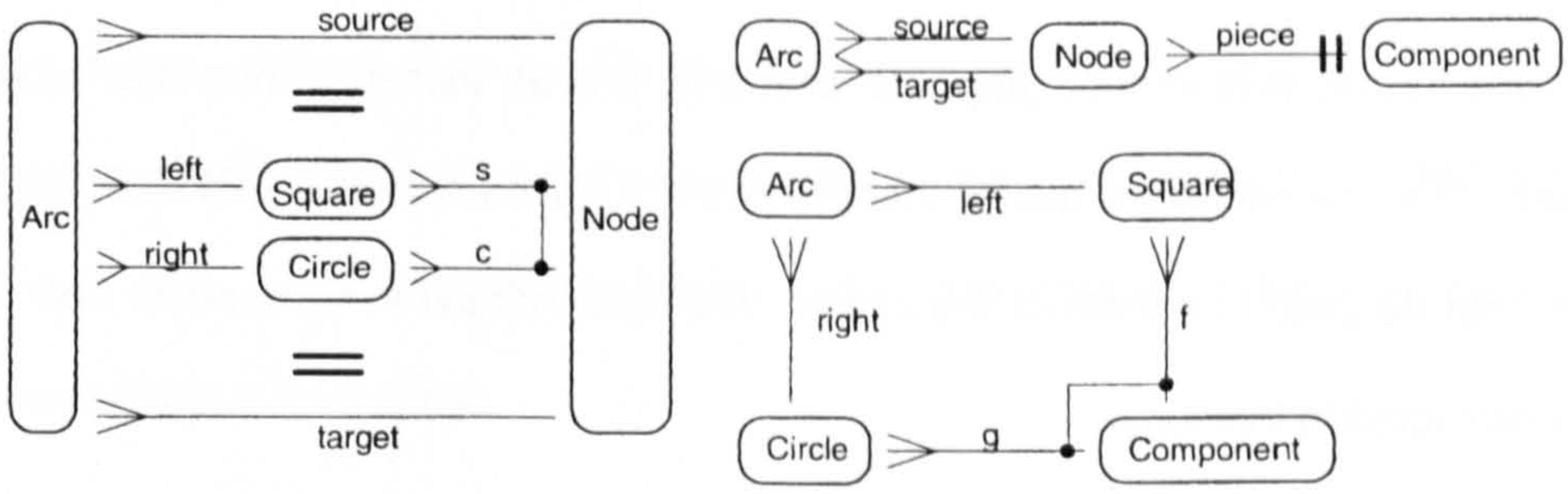


Figure 5.20 Constructing a pushout via a directed graph

The two maps can be determined from the map *piece* in [fig 5.20 upper right] by the equalities:

`f = s;piece` and `g = c;piece`.

5.2.4.2 Restricted Maps

Injective and Surjective functions are represented by placing a single bar on the connector symbol.[14] These kinds of map can be derived by using the previous constructions as constraints. The two examples are taken from the digraph constructions (§5.2.2).

Figure 5.21 Notation for injective and surjective maps

Above [fig 5.5] it was noted that *Loop* is a subset of *Arc*, or in other words the map *arc* is injective. This consequence of the loop construction can be expressed by placing a single bar on the foot of the connector [fig 5.21 left]. The injective property can be defined in terms of a pullback construction which asserts that if two loops are assigned to the same arc, they must be the same loop [fig 5.22 left].

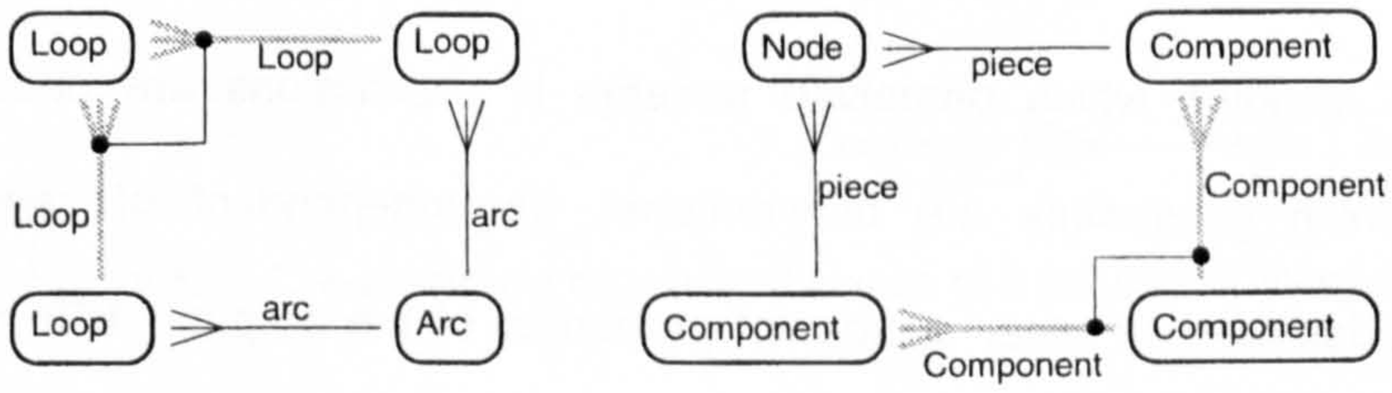


Figure 5.22 Defining injective and surjective maps

The map *piece* which assigns nodes to components is surjective because components cannot be empty; surjection is expressed by putting a single bar at the head of the connector [fig 5.21 right]. The property can be defined by a pushout constraint which asserts that every component has some node assigned to it [fig 5.22 right].

---

[14] Strictly speaking, the constructions define the weaker notions of **monic** and **epic** maps in a general Category.

5.2.4.3 Subsets and Set Operations

An injective map is often used to represent a subset of some pattern of occurrences that is recognized by a further syntactic or graphical property; it is an *inclusion* map that relates part to whole of a set. The intersection of two subsets is simply a pullback of their inclusion maps, while a pushout can then be used to construct the union. We note that set complements and differences cannot be constructed in this logic.

5.3 Building Syntactic Descriptions

The next step is to use the constructions that are depicted in SIGN schemas to build up a syntactic description. To explain how this is done, this section presents basic syntax for several structures that are familiar in many notations, such as trees and textual labels. Sketches for these standard sub-structures serve to demonstrate the strategy and the style of reasoning used, and to provide a prelude to a short study of Jackson Structure Diagrams.

5.3.1 Reasoning about Syntax

The task we consider is that of developing syntax from an informal description of some notation, together with a received corpus of expressions. For this purpose we need a strategy and a method of reasoning.

5.3.1.1 The Strategy

The task entails deciding which perceived patterns in expressions are needed to establish meaning, and which constraints are conventional, i.e. independent of semantic likeliness, pragmatic appropriateness or other accidental properties of the corpus. It is not necessary to define *all* the patterns which could be perceived, nor assert *all* the constraints which are seen to hold, but only those that are conventional or pertinent to meaning.

In defining syntax, the general strategy is to propose a signature of basic entities and maps, then to extend this with further entities and maps or constructed patterns (entities and maps that are defined by certain constraints), and then to state further constraints on all the maps. The extended signature encompasses a wide class of forms. The goal is to make the constraints define a certain subclass: those considered *well-formed*.

5.3.1.2 Deduction in Sketches

This strategy requires an understanding of the properties of constructions and the consequences of applying constraints, and a way to reason about them. The rules of logic available for reasoning formally about syntactic structure are those *internal* to the chosen doctrine. FM-sketches have effectively six kinds of deductive operation, which are classified here in outline as three constructions and three inferences.

Constructions expand a sketch by adding new maps or entities that are implicit in all its models:

C1) A new map may be constructed by using an equality diagram to compose a path of maps;

C2) a new apex entity and connecting maps may be constructed on a suitable base;

C3) canonical maps to or from a constrained apex may be constructed in certain circumstances;

The inferences extend a sketch with equalities that must hold in all its models:

E1) new equalities can be inferred on the faces of cones and co-cones;

E2) new equalities can be inferred from the uniqueness properties of maps belonging to constraints;

E3) new equalities can be inferred from rules for extending equalities.

An example of the rigorous use of these reasoning processes is demonstrated in Appendix C.

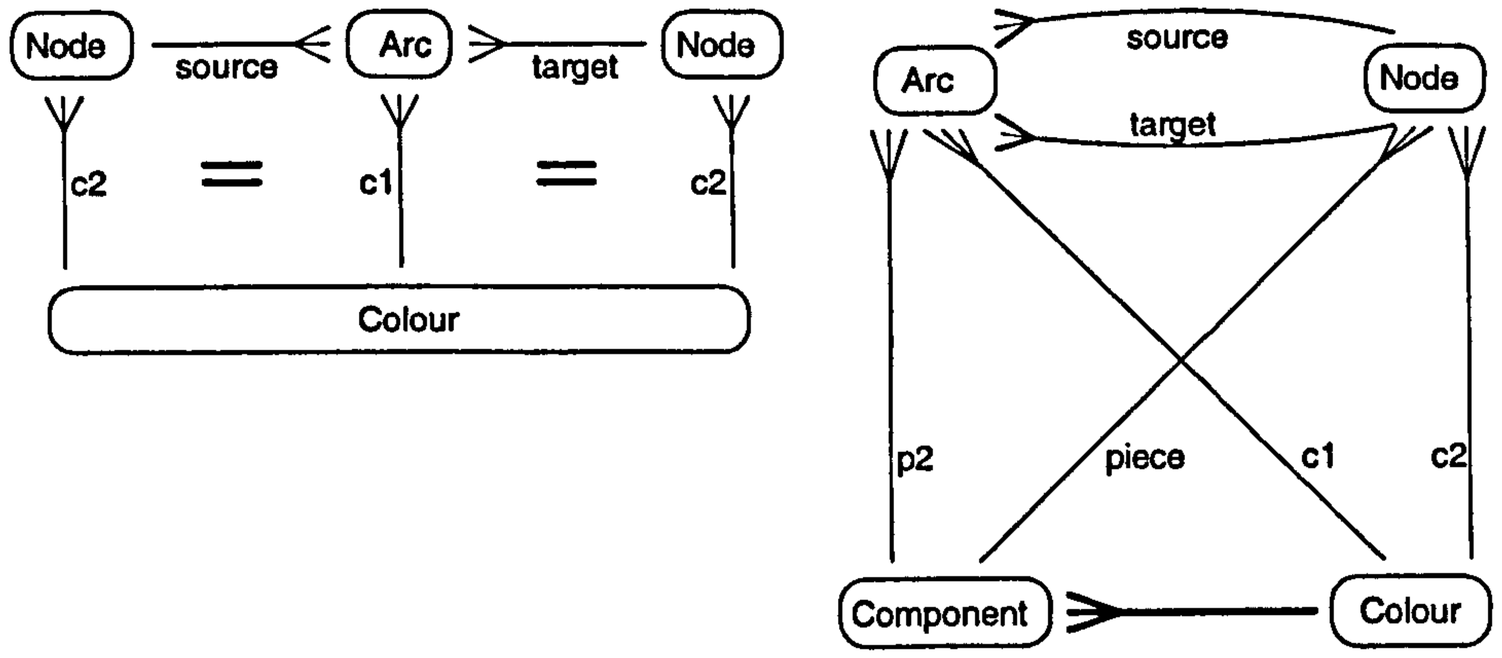


Figure 5.23 Constructing a map from the apex of a colimiting cocone

Several examples of construction of apices, sides and face equalities of (co-)cones have already been given in the preceding section. To give a further illustration, [fig 5.23] shows the cocone constructed as in [fig 5.3] (§5.2.2) to specify components of a graph, and a construction for a map that colours them.

Following rule (E2) the cocone face equalities are:

`piece;source = p2 = piece; target` (implying [fig 5.4]). The schema (left of [fig 5.23]) shows the circumstance where arcs and nodes are coloured by elements of a set called *Colour*, via maps *c1*, *c2*. Two equalities depict the given constraints that all arcs shall have the same colour as their source and target nodes. From these constraints we can deduce that each component is uniform in

colour. The constructed map from the apex *Component* to *Colour*, shown in bold on the right of [fig 5.23] then assigns the colour of each component. The existence and uniqueness of this map is assured by rule (C3) of sketch logic.

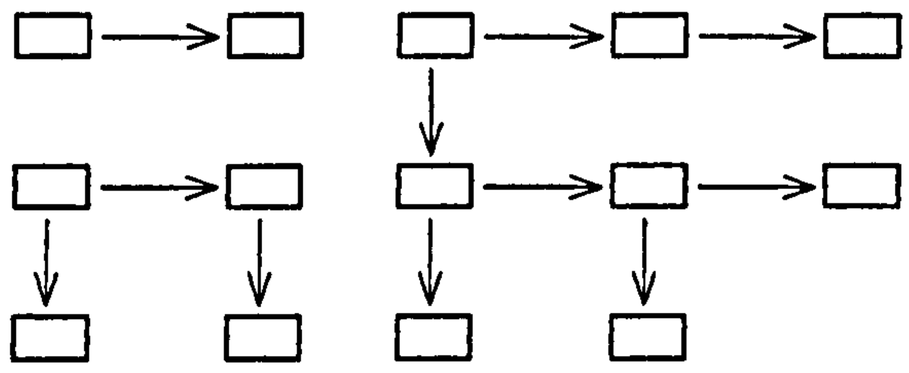


Figure 5.24 An example Forest of Trees

5.3.2 Some Examples of the Strategy

The strategy is illustrated by building a series of example specifications, from trees to alphabetic and textual labels.

5.3.2.1 Trees In a Forest

Our first example is a graph in the shape of a "forest", consisting of "trees" each with a single base node. A forest (e.g. [fig 5.24]) is essentially a digraph (see schema [fig 5.2]) subject to extensions and restrictions, to be expressed by means of extra entities *Tree* and *Forest*, and maps *tree*, *base* and *forest*.

There are three "trees" in [fig 5.24]; each tree has a base which is a node (these are the rectangles at top left, top middle, middle left). There are various restrictions which apply to a digraph which is a forest, for example:-

a) no node is target of more than one arc;

b) every node is either target of some arc, or otherwise the *base* (root) of a tree.

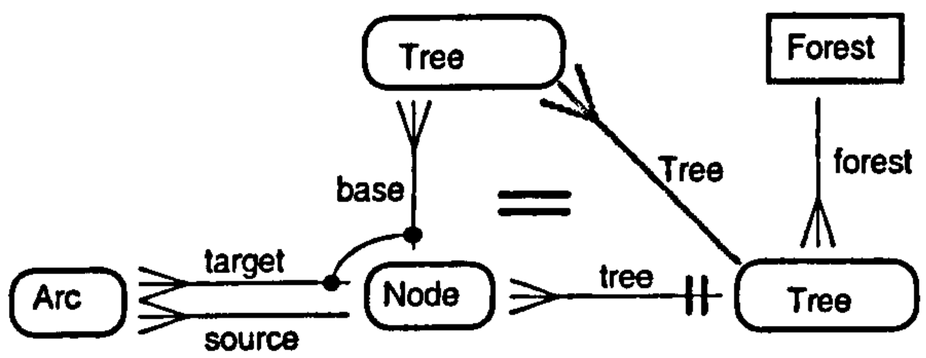


Figure 5.25 A sketch for a Forest of Trees

We can represent these properties indirectly. Evidently trees are the components of the graph (nodes in a tree are connected), so that we may start with a relabelling of [fig 5.3], and extend this into the schema [fig 5.25]. We see that each node of a tree plays one of two roles. The map *base* is intended to select those nodes which are <u>not</u> the target of any arc. It is not canonical, and

therefore the sketch asserts its *existence*. The syntax describes a digraph, its components, and an association of each component with a base node.

The schema [fig 5.25] shows the trees [*Tree*] defined as components. A node is either a base or a target, but not both. Thus the entity *Node* denotes the disjoint union of base-nodes and target-nodes, as expressed [fig 5.25] by linking the maps *target* and *base* at their heads. The map *Tree* denotes the identity function on the set of trees, conventionally named after its entity (§5.2.3). The equality: `(Tree = base; tree)` expresses the property that each base-node belongs to its own *tree*.

The whole forest

All the trees constitute a single forest, so a further constraint makes *Forest* denote a singleton set; this property is signified by rectangular corners on the entity-box.

The forest formed by the trees is identified as an unique item. Generally, for any expression, the expression itself, regarded as a whole, is an example of a singleton; every item of any sort in the expression belongs to the whole.

5.3.2.2 Sequence

Next we look at a way to define a sequential arrangement of arcs and nodes.

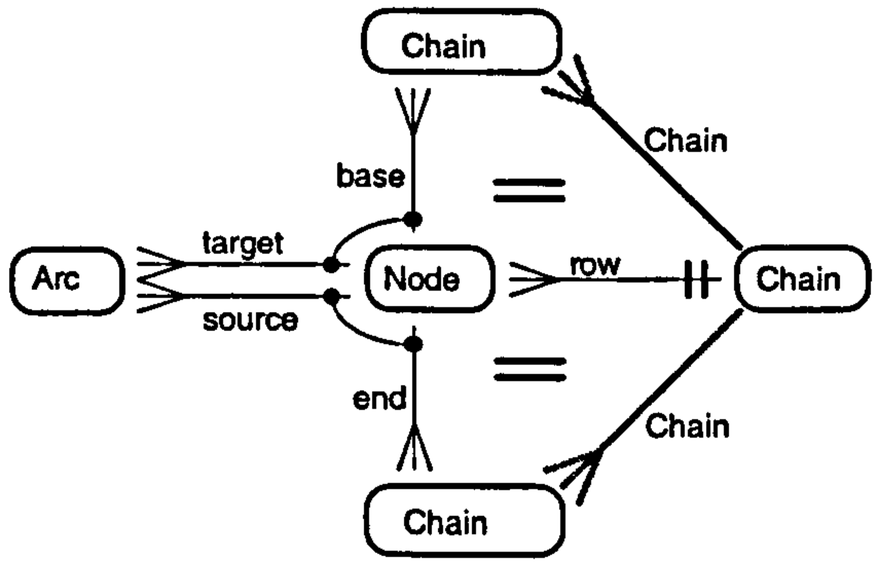


Figure 5.26 A sketch for Chains

The method of description here makes use of the symmetry inherent in sequence: that the reversal of all arcs preserves the sequential property. By contrast, an 'inverted' tree (i.e. with arcs reversed) is not normally still tree-shaped.

Observing that source and target maps of a sequence must therefore be constrained in the same way, it is simple to add a further map to the sketch for tree and make the sketch symmetric. In this case the symmetry of structure can be translated into a symmetric layout for a schema.

Chains

A *chain* is a sequence of nodes, and can be seen as a tree which cannot branch. ·Applying the same constraint to the leaves of the tree as to its base yields the syntax for a chain [fig 5.26]. The *base* of a chain is its starting node.

Note that an empty chain consists of just a single node, its *base*, which is also its *end*.

5.3.2.3 Alphabetic and Textual Labels

Building on the above, we next investigate structural aspects of words, viewed as graphical expressions. The description does not concern the language in which the text is written, nor the layout on the page. It treats textual strings as sequential graphs whose arcs are alphabetic tokens.

Although in many notations characters and text strings play a special role as lexical items (e.g. keywords), they can also appear as labels, for example in quantified formulae, in program code or on diagrams. In order to describe the structure of labels, we need to define equivalence classes and relations.

5.3.2.4 Alphabetic Tokens

Firstly an *alphabet* is defined as a set of distinguishable characters. Where the alphabet is a fixed set of significant shape-types, printed as a set of tokens, these token-sets can be represented as character entities in the syntax.

> As with all the items discussed so far, character tokens are shapes with a definite location on a diagram; thus each character has its own entity (*TokenA*, *TokenB*, etc.) on the schema. The entity *Alpha* stands for all character tokens together, a disjoint union of all token-sets [fig 5.27 left].

In order to record when two tokens have the same shape, we must define an *equivalence relation* on them. This relation can be treated standardly as a digraph whose edges link related items – but we note that these links are perceptual, and not drawn. The relation can be constructed from the character-entities:-

> The schema [fig 5.27] expresses an alphabet of only four characters. The entity *EqA* represents a set of pairs of tokens, formed as an union of product sets:
> *AA* is TokenA × TokenA, etc.; because tokens are equivalent only if they belong to the same character-token set.

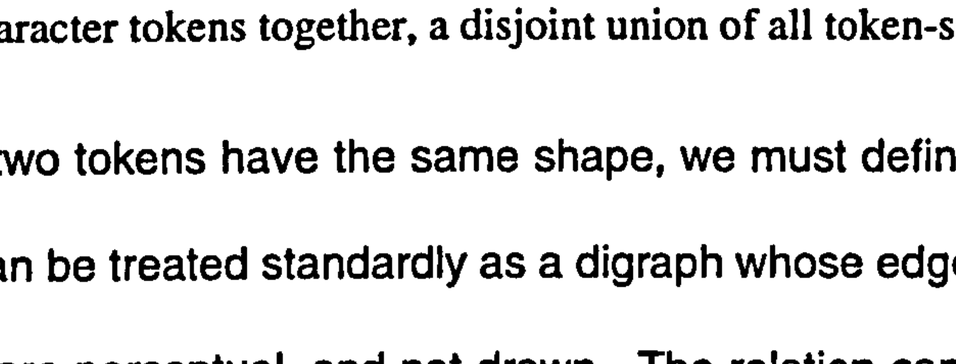


Figure 5.27 Tokens and an equivalence relation

Since all characters that are used as labels have much the same role in the syntax, this

description is somewhat repetitive.

The relation allows us to define a set of shapes, but not an entity for the alphabet itself:-

We can construct from *Alpha* the set *Character* of characters (shapes) in use, as the set of equivalence classes (of character-tokens) [fig 5.28]. These are simply the components of the 'graph' of the equivalence relation *EqA*. The maps *a1* and *a2* are definable respectively from all the first and second projection maps from pairs to tokens, which are seen in the centre of [fig 5.28].

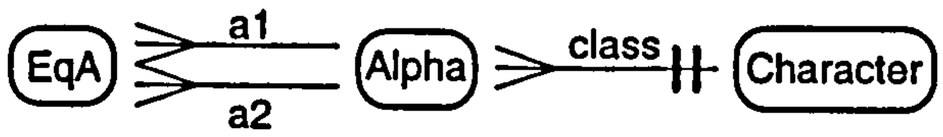


Figure 5.28 Constructing the equivalence classes

It is also possible to "work backwards", by specifying an abstract entity, known to the users of the diagram, denoting a set of shapes.[15] This alphabet of character-types is not a 'concrete' set of items drawn on the diagram. When such abstractions are included in the sketch, the morphisms between expressions do not preserve individual characters – two isomorphic expressions might employ different alphabets. This might be appropriate when characters are variables, labels or other place-holders, but not when they have true lexical roles like the '+' sign in formulae.

Alphabetic Shapes

Suppose the alphabet is represented on the sketch by an abstract entity *Abc*. A map *shape* in effect assigns a shape to each token. The entity EqA can then be constructed canonically as a set of pairs of tokens subject to the condition that the first (a1) and second (a2) tokens of a pair have the same shape. In the schema [fig 5.29] this is notated with a right-angle symbol.

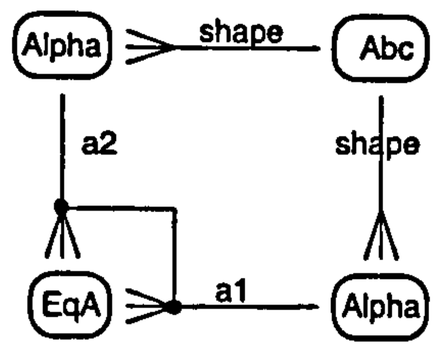


Figure 5.29 Equivalence relation

5.3.2.5 Equal Words

We next turn our attention to the use of words as labels. Words on an alphabet can simply be specified as chains of spaces linked by tokens, each of these being assigned a shape from an alphabet [fig 5.30]. We assume that an expression contains a set of words.

---

[15] The advantage of this is that the alphabet could be extendable (e.g. for other language scripts) without changing the specifying sketch. This is also an illustration of the way that semantic and interpretive concepts in specification of a notation can easily be integrated with the syntactic sketch.

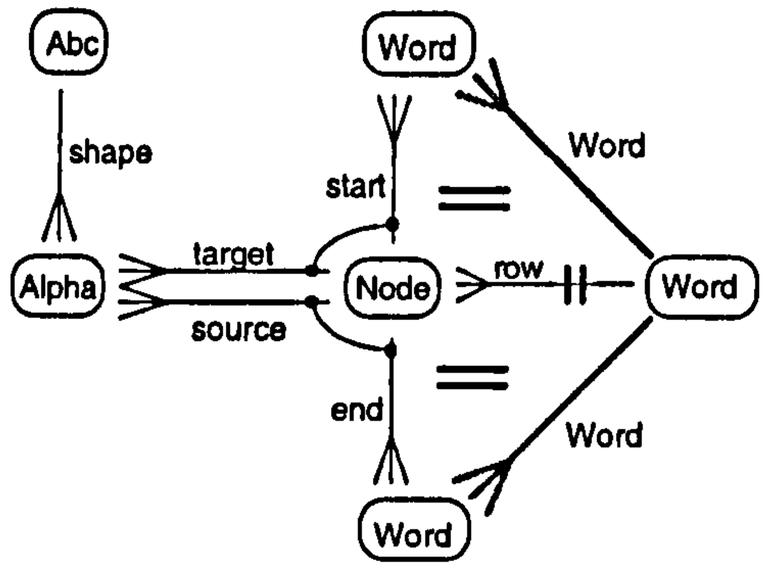


Figure 5.30 The sketch for Words

We consider the set of 'words' (word-tokens) in an expression, and the problem of finding when two words are the same label. It is instructive to sketch a specification for equivalence of two strings, though this bears little relation to the human faculty for recognizing and discriminating words. The demonstrated solution applies the strategy of maximal matching of substrings.

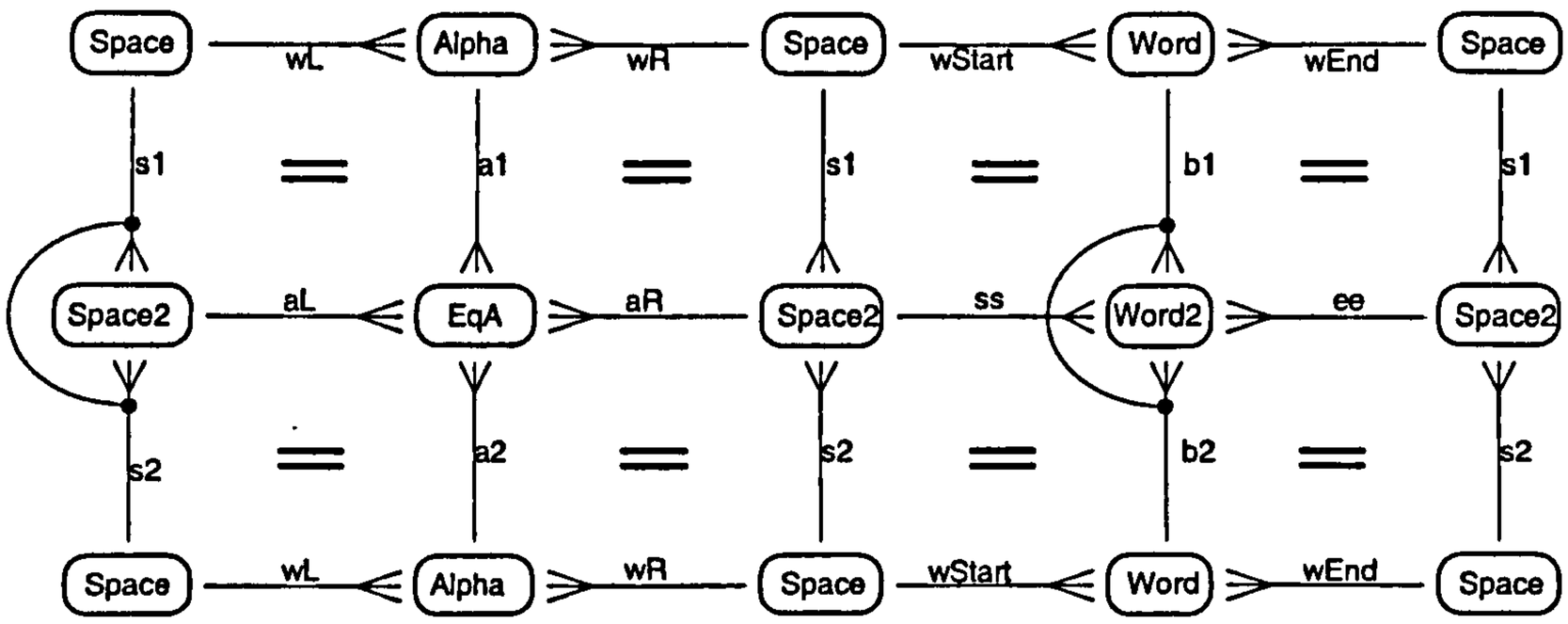


Figure 5.31 Pairing equivalent words

The equivalence relation EqA on the alphabet [fig 5.27] finds every pair (*a1*, *a2*) of identical character-tokens ("alphas"); each must occur in a pair of words (possibly the same one). The next step is to "zip" together subwords along corresponding pairs of spaces. To do this, each of the links EqA given between identical alphas, is used to link the pair of spaces *aL* immediately to the left (in the two words) with the pair of spaces *aR* immediately to the right of both alphas [fig 5.31 left]. *Space2* denotes pairs of spaces, and the maps *aL* and *aR* are here uniquely defined (in the logic of sketches) from the other maps shown.

The right side of [fig 5.31] finds, for any pair of words, the pair of spaces *ss* which start their names and the pair *ee* which end them. These end-points can then be used to find which pairs of words have been fully zipped together.

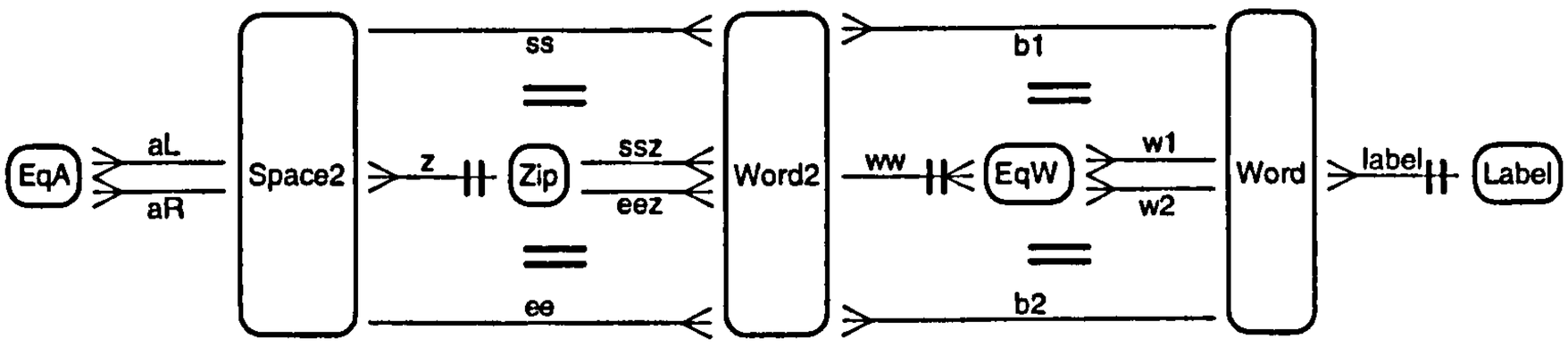


Figure 5.32 Zipping like words together, to give labels.

The parallel maps *aL* and *aR*, regarded as a graph, yield a set of components *Zip* which are the maximal matched subword pairs [fig 5.32 left] – because components of a graph are maximally connected. Then each word-pair in *Word2* has a zip *ssz* which starts on *ss* and a zip *eez* which ends on *ee*; if these are the same zip, then the word-pair is fully matched. Hence equivalent pairs *Eqw* of words can now be constructed as loops in the 'graph' *ssz* & *eez*. Each equivalence link attaches a word *w1* to a word *w2* (sometimes the same word) [fig 5.32 right].

The equivalence graph *w1* & *w2* on words has components which are all the labels used in the expression.

We see in this example that the schemas describe what amounts to an abstract computation within syntax.

5.3.3 Jackson Structure Diagrams

We now have enough groundwork to present a syntactic specification for Structure Diagrams of the Jackson (1983) method of software development. The schemas below are based upon Jackson's exposition, where restrictions on syntax are on the whole carefully stated; in some cases there is looseness in his description, necessitating reasonable assumptions to be made about to what is intended.

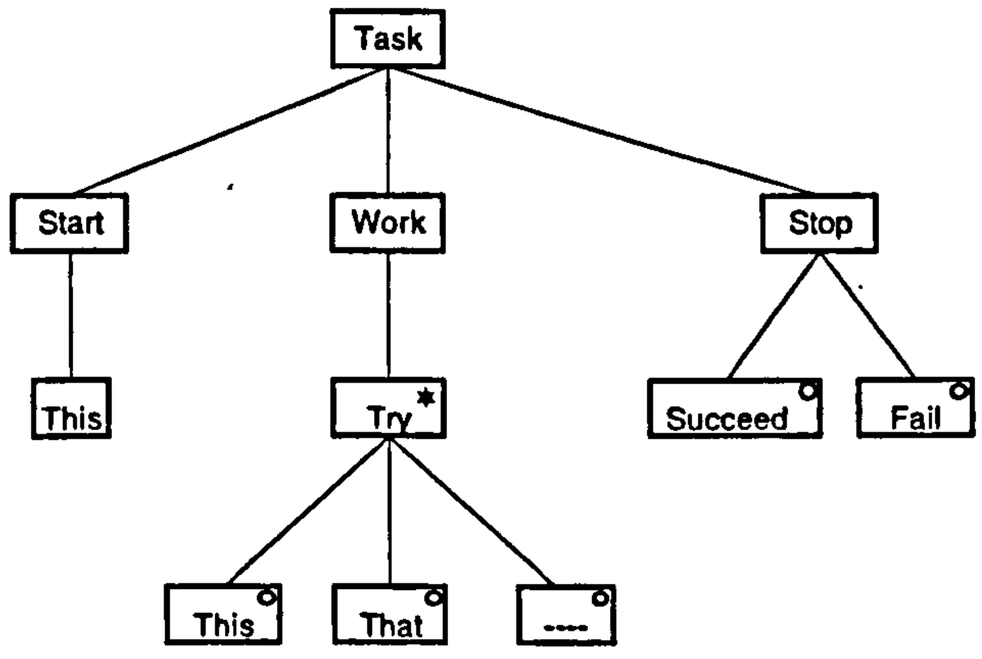


Figure 5.33 An example Jackson Structure Diagram

Structure diagrams are intended to express the analysis of a process as a sequence of actions generated by a regular grammar. Each diagram displays in effect a parse tree for a regular expression. Syntactically, each diagram (e.g. [fig 5.33]) is apparently a kind of ordered tree structure with labelled nodes; it is this structure which are now described.

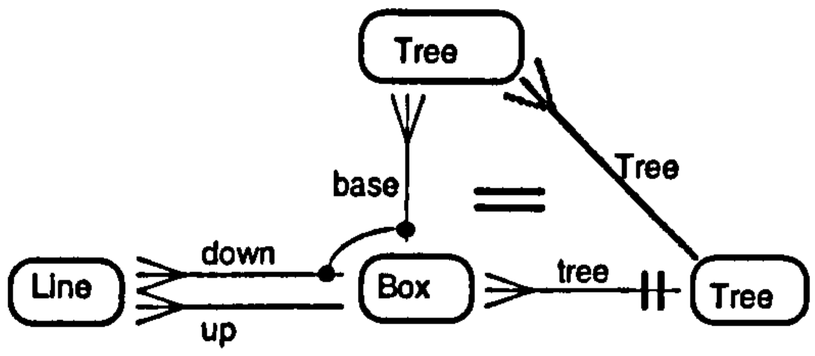


Figure 5.34 A Tree structure

Trees can be specified as in the preceding section [fig 5.24], with some name-changes. [Fig 5.34] allows for a diagram (or document) to contain several trees, together treated as a graph whose nodes are the (process- or action-) *boxes*, and arcs appear as *lines* joining boxes. Each line thus connects two boxes: *up* and *down* (sometimes called "parent and child"). Each box belongs to a *tree* that is a component of this graph. If a box is uppermost (i.e. not *down*), it is the *base* of its own *tree*. The schema depicts these facts.

5.3.3.1 Ordered Trees

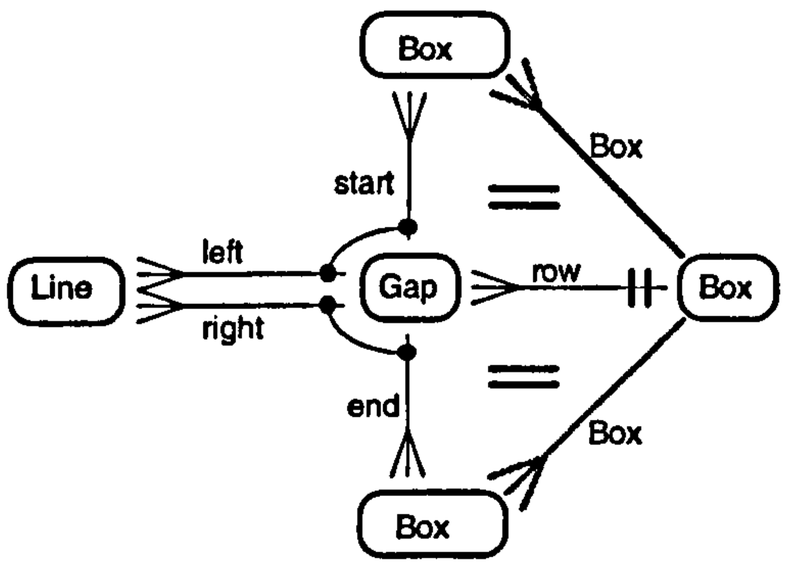


Figure 5.35 A row of lines below a box

The notion of *Chain* [fig 5.25] can now be borrowed to provide an ordering on the lines below a box. Schema [fig 5.35] defines a second graph structure in which lines join a series of *gaps*: empty regions beside a line or between neighbouring lines. Each component of the graph is a *row* (of gaps) corresponding to exactly one ('parent') box; in this case the construction acts as a constraint on the entities *Line* and *Box* that are in common with [fig 5.34].

The component property ensures that for each line, `(left;row)` and `(right;row)` are the same box; in the diagram we see that this is also the 'parent' *up*, as depicted by the equality on the right side of [fig 5.36].

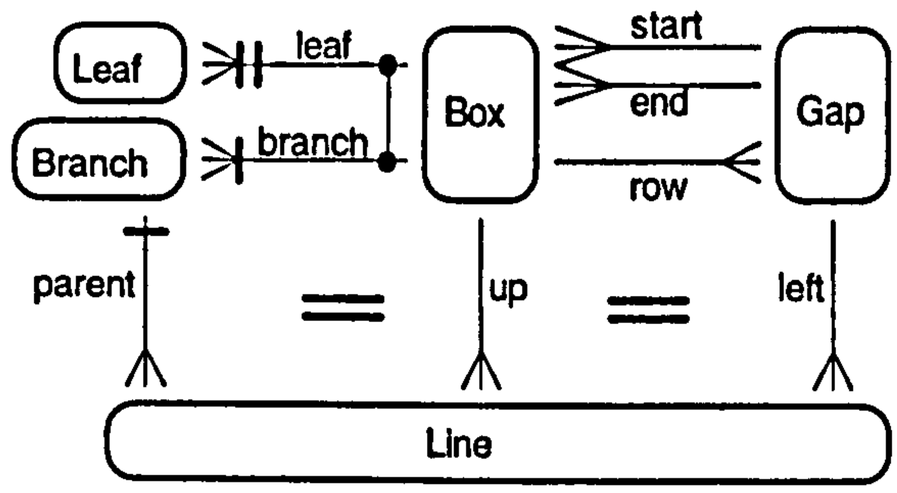


Figure 5.36 Leaves and branches

It is necessary to distinguish two kinds of boxes: leaves and branches, because they have different syntactic properties. By means of a third graph, the top left of the schema constructs *leaves* as boxes which have no children: each box is regarded as a connecting arc between two gaps, namely those which *start* and *end* its row of lines. If this row is empty, its start and end coincide, so that a leaf is a loop of this graph (see [fig 5.5]).

A *branch* is any box which has lines below it: every branch is a *parent* of some line (this map is marked as a surjection). *Box* is then the disjoint union of *branch* and *leaf*: either a box has children or it does not. Further, the map `(parent;branch)` identifies the 'parent' *up* from the line.

5.3.3.2 Sequence, Choice and Iteration

Branches (unlike leaves) are of three types, according to how their dependent boxes ("children") are marked. They are *Sequence*, *Choice*, or *Iterator*; further, an iterated child must be an "only child". These three types are recognised by inspecting their children: Children are marked either *Blank*, *Circle*, or *Star* (though "blank" is actually the absence of a mark). Hence *Line* and *Branch* are both constrained as disjoint unions [fig 5.37]. The correspondence between types (of branch and line) is shown by "right-angle" links between *sequence* and *norm*, also *choice* and *circ*; this construction ensures for instance that *sequence* is derived by "pulling back" the map *parent* along *seq*, i.e. restricting *parent* to the subset *Blank*. Blank can be seen as a relation between lines and sequence-processes.[16]

[16] It is interesting that Jackson chooses to mark the row of dependent boxes to distinguish the three kinds of process, rather than the single parent boxes.

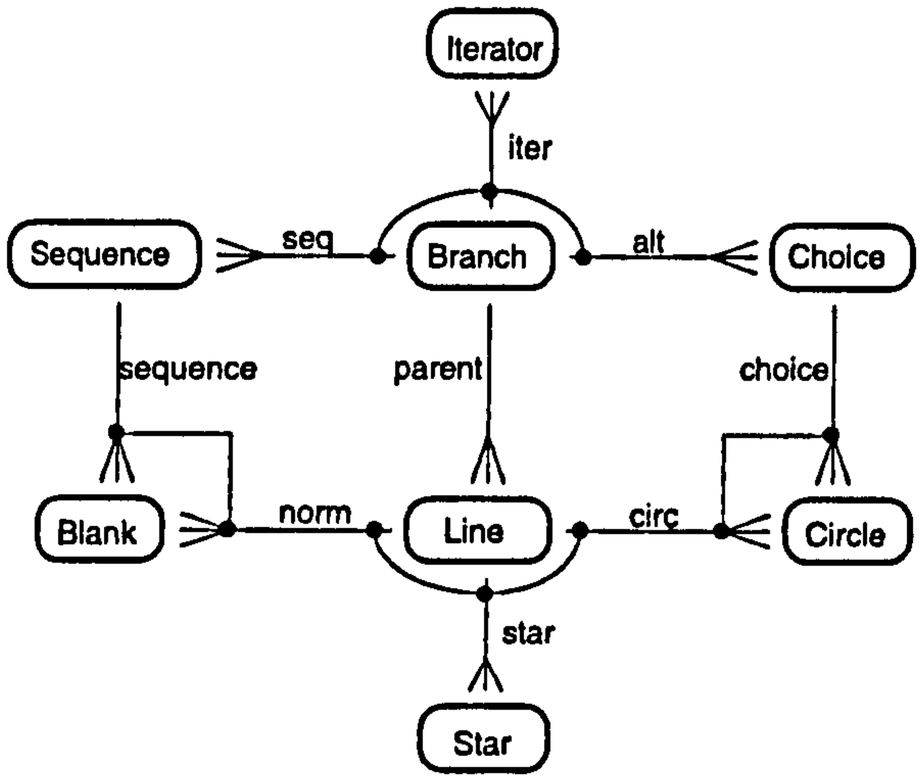


Figure 5.37 Three types of branch

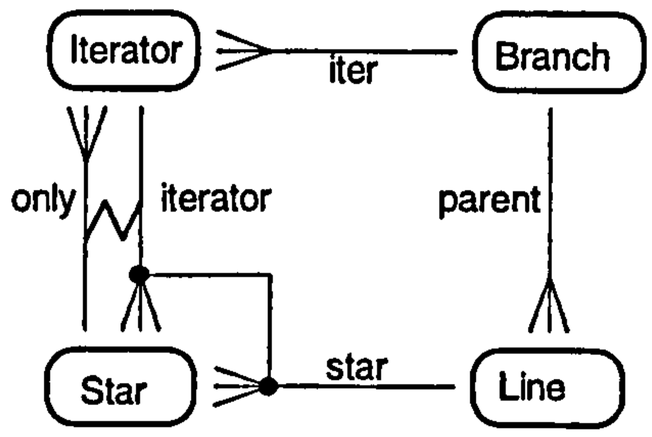


Figure 5.38 An only star-child

The entities *Iterator* and *Star* are isomorphic; the bijection between the sets they denote is formed from the maps *iterator* and *only* [fig 5.39], which express the fact that iterator processes have just a single starred child.

Jackson states the further restriction that a choice process cannot have only one "alternative" (though a sequence may have only one child); though this is not notated here, it would present no problem to do so. Constraints of this kind, which are motivated by semantics, may be included in syntax or not as desired.

5.3.3.3 Names

This short study of Jackson's diagrams concludes by expressing restrictions on naming of boxes, which are not fully stated by Jackson. A name in a branching box refers to a *process* which is specified in the subtree that it subtends, whereas a name in a leaf box refers to some *action* of the system that the diagram describes. In order that a process does not receive two different specifications, a syntactic rule may be added to ensure that each process-name is used only once. Thus names for process (branch) boxes must be unique, though leaves may share names (both within and between trees).

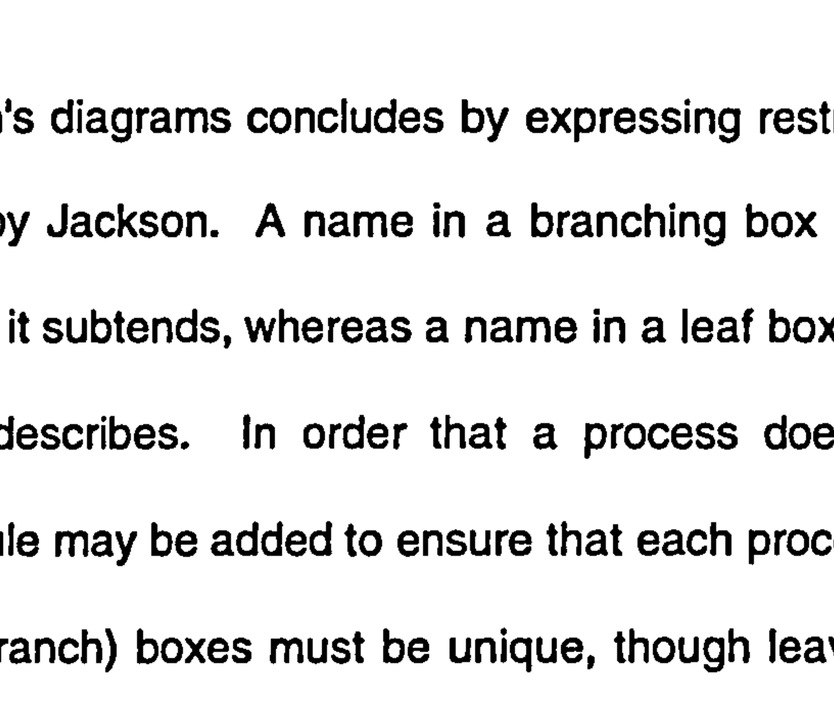


Figure 5.39 Words in boxes

Each box is constrained to correspond to a string of alphabet tokens from a set denoted by *Alpha* [fig 5.39 top]. Names can be specified as labels consisting of character-strings, as explored above (§5.3.2). By two schemas similar to those given there [figs 5.31, 5.32], names may be constructed as classes of equivalent strings.

> Every box contains a name-token, which belongs to some equivalence class in *Name*. The lower right square of the schema [fig 5.40] indicates that the assignment of boxes to names must be unique when restricted to processes, and the upper right square constructs the set of action-names, which may not also be process-names.

There is a special name for the null action, drawn as a dash. The null name can conveniently be identified with the empty word, which consists only of the space at its start and end [fig 5.39 bottom].

> *Null* denotes a set of identical dashes which are placed in boxes which do not contain a word. Only leaves can have the null name [fig 5.40]. The marks on the maps *null* and *nullcirc* signify injections: at most one dash may be written below a circle, and at most one dash may be written in a leaf-box.

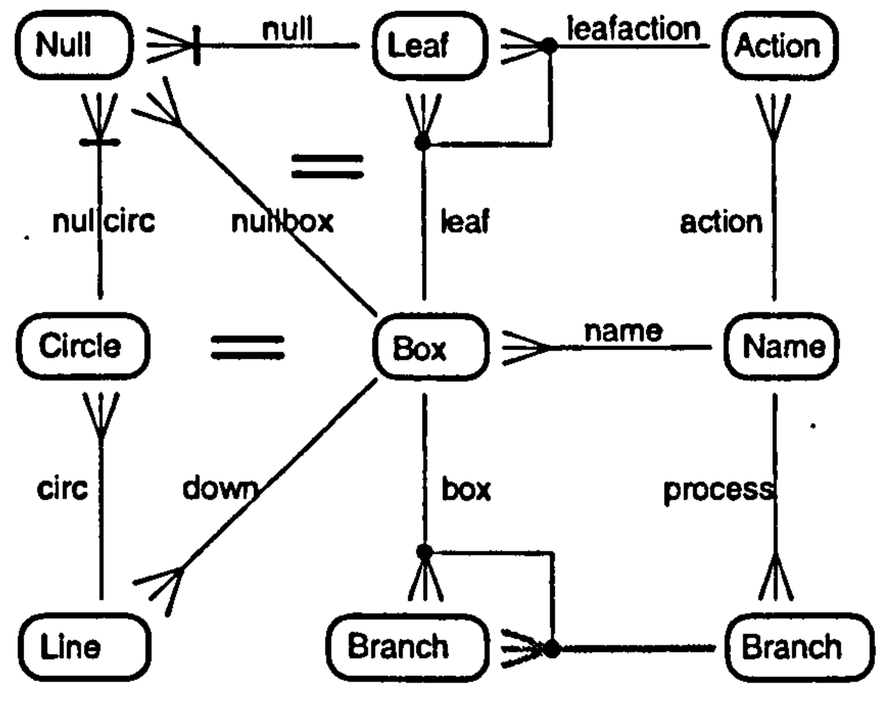


Figure 5.40 Names

Jackson states that a null process is not allowed in sequential or iterated leaves, while at most one null is permitted in a choice. This latter is not notated here. Also not considered here are cases where names have internal structure (with parenthesized arguments), that are found in some of Jackson's examples.

5.4 Discussion of SIGN Design Issues

After the work of explaining how SIGN, as a sketch-based formalism, can provide a graphical way of defining syntax, the last task of this chapter is to examine the design details of SIGN itself. This section describes and assesses the pictorial, syntactic and logical aspects of the notation, and

suggests extensions or alternatives that might improve its suitability or usefulness. To end with, a brief summary is given of the chapter's work.

5.4.1 An Assessment of SIGN

The version of SIGN demonstrated above is intended in the first instance as a vehicle for research rather than for general use; it aims at simplicity rather than sophistication. We first look at the features chosen to achieve this aim, and assess their adequacy in the context of SIGN's role as a vehicle for reasoning about the design of notation syntax. As was noted in Chapter 2, the mere use of a diagrammatic mode in a notation does not justify a claim that it is more beneficial than a textual version. According to the analysis of (§4.1.3), we need to assess how engaging, instructive and communicative it succeeds in being.

5.4.1.1 Features of SIGN

The design for SIGN has a clear origin; it is derived from entity-relation diagrams and category-theory diagrams, which are each known to be useful both informally and formally. In order to keep the rules for reasoning about syntax simple, SIGN is based on just a small number of primitive notions.

Here four specific features and conventions are given that are expected to make SIGN simple and effective. By graphical means, these features help to reduce visual complexity without introducing ambiguity.

1) The (indexical) use of textual names provides a cross-reference mechanism within and between schemas, which makes it possible to simplify layout and avoid overly long or confusing connectors. Names also offer an informal verbal interface with accompanying natural language narrative.

2) Since schemas may then be drawn as plane graphs (i.e. without crossing connectors), the properties of plane geometry can be exploited. Thus the polygonal *regions* that are bounded by paths are helpful in making constraint-symbols concise; for instance, an equality-sign relates to the two 'parallel' paths that bound the immediate region that holds it.

3) Geometric arrangements are used: e.g. loop (and component) constructions use geometrically parallel alignments of connectors in order to bind the syntactic construction together.

4) Some weak iconism is applied: the shape and placing of certain symbols is chosen to help cue their logical properties. For example, the (unconventional) use of a bar as marker for both injective and surjective maps tells viewers that these restrictions are mutually dual; also, the single bar denotes a logically weaker constraint than that implied by the double-bars of 'loop' and 'component' maps (§5.2.4).

5.4.1.2 Modularity of SIGN

As a result of feature (1), flexibility is permitted in expressing sketches in a modular fashion. A schema can be thought of as a frame of reference or viewpoint that highlights certain formal relationships and hides others. It may correspond to either a sentence or a paragraph in natural language. A schema 'sentence' collects together entities that form some important substructure, such as the Tree of [fig 5.34], whereas a 'paragraph' may tell a story of how several substructures are combined, as in [fig 5.36].

In many ways, we see that schemas perform a similar function to schemas in *Z* notation – but with some important syntactic differences. In *Z*, the signature and explicit predicates (i.e. constraints) are placed separately within a schema box, whereas SIGN mingles them. We see that SIGN employs much less abstraction than *Z*; SIGN schemas are not arranged in a decomposition hierarchy. For example in SIGN, a schema is not named, and thus cannot be referred to within another schema, as is possible in *Z*. Also, SIGN has no mechanism for re-naming entities in a schema or for formally combining schemas. In the next chapter a further notation for depicting relations between sketches is suggested, which goes some way towards filling these gaps.

5.4.1.3 Adequacy of SIGN: Conciseness and Richness

SIGN is proposed as a working conceptual tool to assist precise design of notational syntax (whether for new or existing notation). How well would it fulfil this function?

SIGN is not designed to be very concise. Generally we would expect diagrammatic depiction of syntax to take more space than any textual equivalent; also, formal description tends to be lengthy because it must make every detail explicit. Taking this into account, the schemas required for the example specification in this chapter are fairly economical, but this could be improved upon, with a more richly expressive syntax, as considered below.

SIGN is deliberately not a rich notation; as it stands, it is too restricted from a practical point of view. The symbols introduced in (§5.2) notate only an elementary set of constructions – almost a

minimal logical basis for sketches – as a result of a particular compromise between convenience and logical generality.

One problem with the reliance on such a small number of primitive notions is that it leads to less succinct schemas. Although the choice makes it feasible to notate constraints *in situ* upon schemas, this carries the disadvantage that names are required for unimportant intermediate maps. The task undertaken to produce the specification above has shown that choosing layout and naming requires care if schemas are to be clear and easy to understand. Each of the many maps must be named uniquely – for instance subsets are named as inclusion maps. Finding suitably intuitive names for the many entities and maps has proven to be an awkward task.

5.4.1.4 Specificity

Do schemas exhibit the property of specificity (§4.3.1) that is needed to help reasoning? Since specificity is determined by the directness of the relation between graphics and semantics, the answer to this depends on what we take to be the semantics for SIGN. Schemas denote sketches, but sketches denote classes of models. Further meanings involve the deductive dependencies revealed by schemas.

If we regard sketch structure as a semantic level within SIGN's syntax, then there is a fairly direct relationship between items drawn in schemas and items of a sketch, though this is subject to the restrictions of feature (1) – several boxes with the same name refer to a single entity.

At another semantic level, a depicted sketch denotes a definition of its models. Though the relationship between sketch and model is not direct, schemas, on the other hand, can be drawn to express models directly. This is achieved by using the syntactic signature as basis for a kind of *proto-notation*, equivalent to a directed graph labelled with names of entities and maps. This possibility arises by virtue of feature (1), which gives SIGN enough flexibility to 'explode' a schema into broadly the 'same shape' as a sample expression in the notation it describes – a technique that can be useful for analysing expression-instances during notation design.

As illustration, we take a proto-notated version of the JSD diagram [fig 5.32] of (§5.3.3); this is derived from the tree schema [fig 5.33], by 'exploding' into schema [fig 5.41] – though note that some connectors are omitted to avoid any cross-overs.

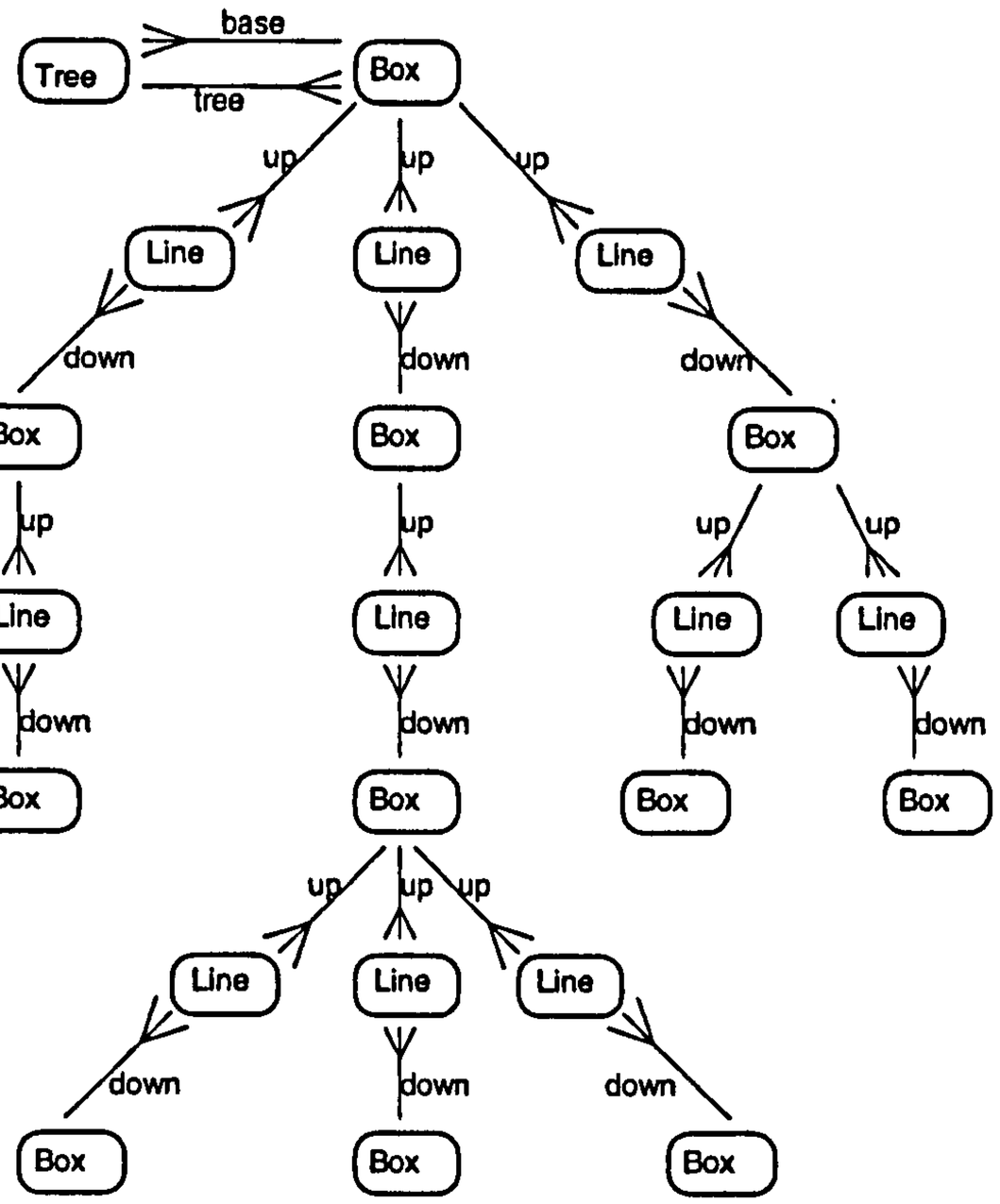


Figure 5.41 An Exploded Schema

As far as semantics for *reasoning* with sketches is concerned, deductive relations within or between schemas are not depicted, but are only accessible via formal logical rules. Though schemas cannot notate deduction, they are important as a visual aid in support of reasoning. The instructive use of SIGN for tractable reasoning about syntactic structure is aided by directness, but as we have just seen, this depends on the level (or fragment) of semantics which is the focus of reasoning. Either different variants of SIGN would be needed, or more flexibility could allow expressions to be tailored to a particular focus.

5.4.2 Redesigning SIGN for General Work

How can SIGN be extended to overcome the above problems? What changes are needed to SIGN to render it suitable for a non-specialist in notation, who wishes to see an instructive formal definition of a standard notation? We look next at ways that expressiveness and engaging qualities of SIGN might be improved – in order to create a more expressive working version without losing its basis in Sketch Theory.

5.4.2.1 Improving Expressiveness of SIGN

Methods of enriching syntax or semantics of SIGN could be applied, to adapt it to both context and practice of design.

In specification notations generally, it is desirable to be able to express as directly as possible those concepts that predominate in the subject domain. Directness is achievable by giving these concepts special symbols, denoting either *primitive* or *derived* elements of the logical formalism. Extending a notation with new symbols can therefore be done in two distinct ways:

> *Syntactic* extension introduces a symbol as a shorthand to replace an arranged group of symbols, without otherwise changing deductive rules. The new syntax can be concrete (replacing a specific group) or abstract (replacing a type of pattern).
>
> *Semantic* extension enriches a notation by enlarging the set of primitives, which entails the reframing of deductive rules, and may cause radical changes that require further mathematical analysis.

Syntactic extension is similar to the technique of *definition* in a logical calculus. Rather than devising a logic with a rich set of primitives, a method of extending the logic with derived notions is preferable to the second option, because it does not require a complete reworking of the deductive rules.

Concrete syntactic extension was employed in the definitions of SIGN symbols for bijection (§5.2.3) and pullback (§5.2.4). The definitional method takes some common sketch and represents it by a special symbol with its own syntax and derived rules of logic. The sketch binds together and constrains several maps and entities; it can be viewed as a *compound* construction, in which some of the maps may have an *auxiliary* function. Auxiliary maps appear only as supporting framework, and will be hidden when a symbol for the compound is defined. Thus in notating the compound construction *pullback*, SIGN hides one entity and three maps (*Arc*, *arc*, *source*, *target*) that are auxiliary to its definition [fig 5.17]. Without a symbol for the pullback, schemas like [fig 5.37] would be far more complex, with many unimportant maps needing to be named.

What is involved here is a compromise between concrete and abstract modes of expression. By adding extra defined symbols, schemas built from commonly occurring compounds can be simplified. Another way to make schemas more concise is to notate general *kinds* of construction[17], to avoid an ever increasing symbol-set. Thus more general derived constructions could be notated, but this is achieved at the cost of increased abstraction and complexity.

As a form of abstraction, we could include syntactic definitional mechanisms in SIGN itself. Such

---

[17]This refers to sketch cones and cocones of any size, i.e. general limits and colimits in categories. It is possible to go even further and notate left and right Kan Extensions, or Ends and Coends (MacLane 1971).

abstract syntactic extensions would involve more complex graphical reasoning patterns, increasing the requirement for professional expertise.

5.4.2.2 Ways to Improve Engagement

What would make SIGN more engaging and communicative? One way is to provide more flexible ways of naming maps and entities.

The conventions for naming of maps could be improved. It may help to develop conventions where maps are given compound names, so that map-labels need not be unique. It should be easy to create a formulaic notation for naming, because categories and sketches determine an *internal language* in which maps are expressed by formulae (Lambek & Scott 1986). The syntax for these internal formulae can be constructed from the doctrinal rules, and manipulated through an 'essentially algebraic' (equational) reasoning process. An experimental method of naming paths is used in Appendix B.[18]

SIGN uses verbal text to label each syntactic entity (box). We could instead or in addition use pictorial cues taken from the target notation, to help the user identify the graphical pattern denoted by an entity in a schema. For instance, an entity named 'arrow' could be illustrated by drawing a typical arrow-icon in the entity-box. In semiotic terms (§4.3), this amounts to an iconic temporary extension to SIGN, specific to the particular target notation. This could be accommodated as an informal annotation, but is hard to see how it could be done formally. In practice, such annotation or extension to SIGN could only be made with the help of a versatile notation-development tool such as that proposed in Chapter 7.

5.4.3 Alternatives to Sketch Theory

SIGN attempts to achieve clarity and avoid ambiguity in its schemas through the support of formal logic, in the form of sketches. Here we consider changing the semantics of SIGN, by using other possible logical bases, and we ask how the method of sketches compares with other formal approaches.

Is a categorical foundation is the most appropriate? The main alternative formal description

---

[18] A simple way to name maps is to prefix a map-label by its domain name, and to name an unlabelled map by default after its codomain.

methods would be the general purpose languages of classical Predicate Calculus (first or second order) and Set Theory. Both satisfy criteria of good mathematical support, but reasoning in these systems is difficult even for experts. As with sketches, there is a hierarchy of strengths for classical logical formulae, that can be used to control the complexity of semiotic relations. Set theory might appeal to a wider mathematical audience than category theory, but neither it nor classical logic are commonly used by software developers.

If having a graphically expressed formalism is important, then sketches are better placed than set theory, for which no such notation has been developed. However, an extension of sketches to accommodate Set Theory could be based on Topos Theory and higher-order logic (Lambek & Scott 1986). Peirce's existential graphs do provide classical logic with a graphical notation that is worthy of investigation. A pictorial first-order logic based on the work of C.S. Peirce (Hartshorne& Weiss 1933) is demonstrated in the conceptual graphs of (Sowa 1984) see (§2.1.1, 2.2.2).

We require evidence that reasoning by graphical constructions and equations in sketches will prove any easier to support than other formulations of logic. The arguments of Chapter 4 suggest that this will only be so if the sketches used are restricted in logical complexity, and if the schema notation is adapted to reflect the restrictions graphically.

We have seen that sketches, through their connexion with category theory, provide for mappings between syntactic structures. This promises to be an especially fruitful aspect of the method, in that it allows analysis of analogy and metaphor, making it particularly suitable for describing semiotic structure. Predicate Calculus does not easily allow such mappings (Roisin 1979).

5.4.3.1 Expressing Properties and Relations

Among the Predicate Calculus concepts that the sketches do not express directly, are the fundamental notions of property and relation. Many geometric and semantic connexions found in general notations seem relational rather than functional (e.g. 'near-to'). Sketches represent such relations indirectly as subsets (injective maps into product sets); hence SIGN lacks symbols for many-to-many relations and partial functions. From (§5.4.2) we see that symbols could easily be defined by syntactic extension, if necessary. A relation could be regarded as a 'loose' function that can yield ambiguous values or no value at all.

The alternative would be to take relations as primitive. The concept of a total function must then

be derived as a restricted kind of relation, with the help of a special relation called 'equality', in the standard approach of predicate logic. A lengthy investigation of fundamentals would be needed to base a notion of sketch upon relations. A possible starting point is Peter Freyd & Andre Scedrov's work (1990) on *Allegories* (Broome & Lipton 1994). The foreseen generalizations of sketches (Bagchi & Wells 1994) referred to in the next chapter may suffice for this.

5.4.4 Summary

To finish the chapter, a short summary is given here of its achievements.

In this chapter a graphical notation and formalism for specifying syntax has been described and illustrated with examples. This work has demonstrated that the proposed notation enjoys a mathematical basis, which is found in the Theory of Sketches and Category Theory. The formalism treats syntax as a set of perceived connectivity constraints which are definable without reliance upon the details of realization in a pictorial medium.

An example specification for Jackson Structure Diagrams has been presented, to include the constructions and constraints for all the properties which Jackson documents, apart from some minor omissions that are indicated. The method establishes the principle that syntactic constraints can be diagrammed formally.

The benefits and deficits of the proposed schematic formalism have been analysed according to the work of Chapter 4. As a result, ways of improving its notational design are suggested, with a view to providing more useful and practical developments of the formalism.

Chapter 6

Support for Notation Design and Processing

Abstract

Here we find an investigation into how the proposed 'tectonic' framework can be applied to problems of computer-aided notation processing. Starting with a look at methods of formal deduction and proof within sketches, the discussion proceeds to define a set of notions based upon maps between sketches. These notions are particularly useful for describing reasoning about syntax and operations on expressions. The technique suggested for designing notation structure then involves building a syntactic sketch from a network of sub-sketches and maps.

The theory indicates a method for diagramming the logical relations between sub-sketches by means of 'meta-schemas', which can be used in planning the computational strategies involved in interpretation of expressions. The question of defining the appearance of a *drawn* symbol is also addressed, recognizing the need for universal pictorial theories into which the syntax can be mapped.

In order to discuss a range of operations on expressions, the tasks of editing are next analysed in detail. Editing is seen as the building of a model, guided by a syntactic theory and directed towards the user's semantic goals. In theoretical terms, creation of an expression is described as the instantiation of its syntactic sketch. In a graph grammar approach, guided editing allows an expression to be instantiated gradually, by inserting temporary symbols for syntactic items that will later be replaced by patterns of items. For this to work in general, it is shown that a separate editorial syntax is needed.

We see how the framework of tectonic sketches constrains a flow of change within syntactic structure, resulting when a graphical expression is modified. Graph rewriting techniques offer an elementary way to implement this flow. These rewriting operations are analysed in relation to logical properties definable within the framework.

Chapter 6.

Support for Notation Design and Processing

The previous two chapters have proposed a theoretical framework and indicated a method for specifying a layered syntactic structure. Our purpose now is to see how this 'tectonic' framework can support techniques for reasoning about structure and operating upon expressions. These techniques are directed both at the designing of effective notation and addressing particular problems of providing computer-assistance. Computer-aided editing is selected as an important topic to address, with a discussion to clarify the processes involved in modifying expressions. A logical analysis is applied in order to resolve difficulties found in design of rewrite-rules for flexible editing methods. Without exploring too deeply, this chapter thus covers a number of areas that pertain to the building of a notation design tool, which will be the subject of the succeeding chapter.

We wish to give assistance to people in several roles: to the technical designer of notations and to those who will employ notations, either as a *viewer* of expressions or as a *notator* who produces expressions.

The activities that users are engaged in come under two headings: employing and editing. Employing expressions involves a range of cognitive actions, more or less tacitly: from creating ideas through to physically expressing them; from perceiving pictures through to impressing their content on the mental state (in readiness to respond). Editing involves some explicit actions which are reliant on this cognition: for instance the building, modifying, formatting and translating of expressions.

There are three areas where the ideas of this thesis may lend support, and three kinds of support that are offered. Problems of *specification* have already been partly addressed; problems of *design* and *processing* are the focus here. *Theoretical* support provides the confidence that *practical* and *computational* support for all areas can be developed. Design of computer-assistance can then follow a policy of providing algorithmic processes that correspond to cognitive ones.

6.0.0.1 Operations

How can we reason about operations? To operate or work upon an expression means to change it in a regulated manner. There are two aspects of this working that require our attention: the structural integrity to be maintained throughout any change, and the means by which the processes of change are to be controlled. As in computing, where these aspects arise as the *declarative* logic of system specification and *interactive* logic of program execution, our problem is how to unify them.

The first aspect involves defining the deductive, semiological processes underlying all notational tasks. In the case of interpretation – the discovery of meaning – semantic facts are deduced from observed graphical properties of expressions. Conversely, in producing graphical forms to express facts in a situation, the task is one of selection or discovery of a display that has the intended interpretation. From an operational perspective, both of these entail constructive proofs-of-existence. As regards the second aspect, we have seen (§3.2, 3.3, 4.4) that the control of change is often achieved by means of rewrite rules. With a view to implementing notational processes, we must therefore examine techniques of graph rewriting in relation to the logical framework.

How do the theories presented apply to parsing and interpretation of drawings, or production and display of expressions – when carried out by computer? This is answered below with the help of the concept of translators between syntactic theories, as introduced in (§4.5.2). Translators of theories would in principle be associated with programs that convert models of one form to those of another, as proposed in (§4.5.2). There it was suggested that interpretive processes encode information in an intermediate abstract form (theory *R* of [fig 4.4]), from which meaning is derived. Here this abstract R-form will be regarded as conceptual data and could be called an *idea*.

Display of semantic data starts by forming an idea. Production of a drawing that expresses it requires a search strategy that selects from a huge variety of solutions, applying informal criteria such as aesthetic heuristics to choose layout.

6.1 Supporting Notation Design

In order to support for the designing of notations, this work must offer help in the logical analysis of notation syntax. Further to the theory outlined in (§5.1), this section elaborates the descriptions of

semiotic structure given in Chapter 4 (§4.5) and proposes language and methods for structural design. The main device to be introduced below is a tectonic sketch for a theory of notations in their context. This overall sketch is resolved into a complex of named sketches connected by logical relationships, which are in turn depicted in *meta-schemas.* The mechanics of the relationship between graphics and semantics is discussed, and finally consideration is given to question of pictorial realization.

6.1.0.1 Various Reasoning Processes

As noted in Chapters 2 and 4, we can discriminate several kinds of reasoning associated with a notation. For the user, the most apparent thoughtful activities occur in the subjective semantic domain: employing an expression to think about possibilities expressed in the subject area (§2.1.2), and potentially complex processing such as the calculation of consequences (which are however outside the scope of this thesis). These activities rest upon transductive capability in interpreting or conversely producing expressions, alluded to in (§4.3). In one direction, interpretive thought extracts semantic properties of an expression from graphical facts; in the other, productive thought expresses semantic properties in graphical form. These capacities rely, in turn, upon an implicit understanding of the notation's structure, which users must develop.

During the design of a notation, these understood processes need to be made explicit; not as cognitive functions, but as formal structure that is related to computational needs. In order to support all these processes, the notation designer must exercise deductive reasoning in deriving systematic properties of a notation from within its specification.

Determining the effect of syntactic constraints involves construction of required maps and entities, and inferring new constraints which are implied by those already asserted. The reasoning method for the sketches is graphically assisted by the SIGN schemas, in an analogous manner to the practice of Euclidean geometry – whereby lines, points and arcs are constructed on a drawing, and new properties are deduced using explicit postulates and theorems.

6.1.1 Theoretical Support for Deduction

We begin with some further details of the theory behind deductive processes, which relies on the construction of a *formal theory* – a category that is the closure of a sketch – which was mentioned in (§5.1.3). The objects of study which a formal theory is intended to describe are approximated

by its formal models.

Both sketches and theories declare the properties of models, but sketches additionally allow us to take some account of 'cognitive effort' in the interpretive processes, inasfar as these are mirrored by deduction on a sketch. By referring to an ideal notion of *theory*, mathematics abstracts away from (or avoids) such concerns. The intention here is to accommodate both of these perspectives on deduction, in a version of logic that is intuitionist rather than classical.

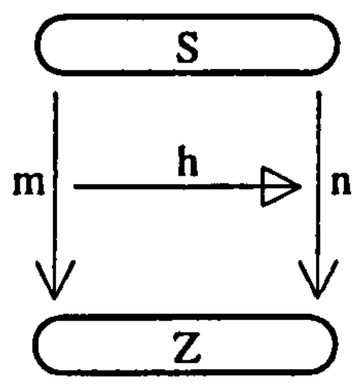


Figure 6.1 A Natural Transformation

6.1.1.1 Formal models

Theories and sketches describe models, in the sense that they identify the set of models that satisfy them, and they establish the systematic relations that hold between models. When a theory is presented by a sketch, we want the models of the sketch to be effectively the same as those of the theory, or of any other sketch that presents the same theory. The category of models of a sketch, referred to in (§5.1.3), will now be defined.

Given a sketch *S* and a suitable category *Z*, the sketch-morphisms from *S* to *Z* are known to form objects of a category `Mod Z(S)` – the category of S-models in *Z*. A morphism between these models [fig 6.1] is an example of a *natural transformation*:-

> If *Z* is a category, a *model* of *S* in *Z* is defined as a sketch-morphism from *S* to *Z*.
>
> A *natural transformation h* between models $m, n: S \to Z$ is a family of arrows in *Z*, indexed by the entities in *S*, such that for all maps $f: x \to y$ in *S*,
>
> $$m(f) \; ; \; h_y = h_x \; ; \; n(f)$$
>
> is an equality in *Z*.

If *Z* is a category of sets and functions, the natural transformation is a family of functions, each of which maps each element of a given sort in *m* onto an element of the same sort in *n*, while preserving the connectivity. This formalizes the part-whole relation discussed in Chapter 4 (§4.3), and for example corresponds exactly to the usual definition of graph homomorphism.

It is usual in category theory to characterize a category by its arrows, and not by any supposed internal structure of its objects. In our case we are especially interested in models that are *concrete* object-representations – arrangements of items, with morphisms that are certain functions between item-sets. The morphisms in this case do not tell the whole story. In particular, we do not here pursue a notion of semantics in which morphisms between graphical models correspond to morphisms between semantic models (§3.2.3).

6.1.1.2 Deduction

In any logical system, deduction is an activity that allows us to develop a theory from a set of postulates. Deduction upon a sketch *S* leads to the extension of *S* into a sketch *T* by adding formally derived entities, maps and constraints – governed by some doctrine *E*. Viewed abstractly, this constitutes a sketch-morphism `d: S → T`, through which every model of *S* is also a model of *T*. The theory generated by *S* is, in a sense, the largest such *T*. On the other hand, the theory is a minimal E-category that includes *S*. It follows that stronger doctrines generate potentially larger theories.

In the following exposition, the general framework for "graph-based logic" (GBL) described by Bagchi & Wells (1994) is the source of formal definitions for these notions. The general notion of a *theory* in a doctrine E can be defined formally in two ways. The first definition yields a notion of *loose* theories:-

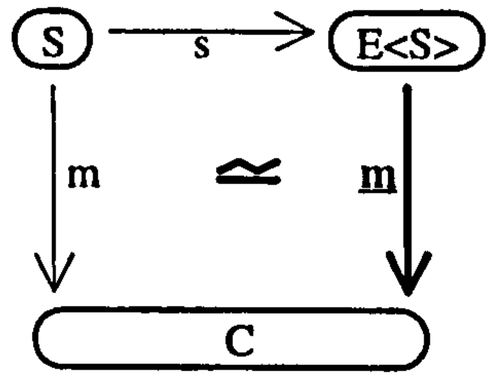


Figure 6.2 Unique factorization of model *m* via the generic model *s*.

Loose theories

If *E* is a doctrine, every E-sketch *S* has an E-theory `E<S>`, which is an E-category together with a sketch-morphism `s`, called the *generic* S-model for *E*:-

`s: S → E<S>`

such that for any E-category *C* and S-model *m* in *C*, there is a model $\underline{m}$ of E<S> in *C*, unique up to isomorphism, for which `m = s;`$\underline{m}$.

Every model of *S* is said to "factor uniquely" through the theory E<S>.

This determines E<S> up to equivalence of categories.

The generic model *s* induces a natural equivalence between `Mod C(S)` and the category of E-functors from E<S> to *C*. In this sense the sketch *S* has the 'same' models as its enclosing theory. The force of the phrase "up to equivalence" is that we do not specify how many isomorphic copies there are of each object of E<S> – i.e. each derived concept. A loose theory captures the notion of a definition that is not tied to a particular presentation *S*.

For the purpose of defining formal deduction it is helpful to have a second way to understand a theory – as a model of a doctrinal FL-sketch. These *strict* theories are E-categories with *designated* E-limits and functors that preserve them.[1] Bagchi & Wells give a set of rules of construction that produce the *strict* E-theory of a sketch as the initial algebra for the FL-sketch of E-categories with constants added describing the sketch (see also Wells 1990).

Strict theories

In the general case a doctrine is established by taking an FL-sketch E and generating its theory $\underline{E}$ (i.e. FL<E>), which is thought of as a *constructor space*. Then $\underline{E}$-categories are just the models of $\underline{E}$ in **Set**. Each object in $\underline{E}$ is a type of construction that can be carried out in any E-category; in a model of $\underline{E}$, each object is mapped onto the set of all possible examples of that particular type.

Technically, an E-sketch *S* is a constant in the limit vertex v of a certain diagram in E, which depicts the signature and constraints of *S*. When this constant `S:1 → v` is formally adjoined to E, it generates a loose theory $\underline{E}$[S] that forces the value of v in a model to contain the signature and constraints of *S*. Then the (strict) E-theory of *S*, E<S> is the initial model of $\underline{E}$[S], a kind of 'deductive closure' of *S*.

The details of this construction need not concern us too much. The effect of it is that an E-sketch *S* allows us to specify any kind of construction that can be made in an E-category such as the medium *Z*.

6.1.1.3 Proofs

The GBL framework also associates a sound and complete proof-theoretic structure with a sketch. In this framework, proof takes place in a *workspace* W in support of a *claim* C about an *hypothesis* H – all objects of $\underline{E}$, that are limits of diagrams in *E*.

An assertion is presented as a *potential factorization* of a map `h: H → W` via a given map `c: C → W`. It is *valid* if it does factorize in every model of $\underline{E}$. The assertion is *deducible* if the claim is verified by existence of an *actual* factorization – a map `d: H → C`, that makes the triangle `(h = d;c)` an equality in $\underline{E}$.

---

[1] The corresponding theory functor is left-adjoint to the monadic functor that treats an E-category as a sketch.

The resulting proof theory is suitable for computer implementation since it makes explicit the full detail of the relationships between different parts of the structure. It has the advantage that proof under any doctrine is always an algebraic operation. For human manipulation, we require a more customary system of inference that is instantiated for the doctrine we are using, and which relies on pattern matching – such as that informally described in (§5.3.1).

6.1.2 Languages for Notational Design

We have seen in the preceding chapter how notation design can be supported by a schematic notation for syntax, which could also be adapted to finer layers of structure. It will now be helpful to invent some further language – appropriate terminology and ideas for new notation – to help apply this theoretical perspective.

6.1.2.1 Translators, Codices and Expressions

In the examples of the previous chapter, syntax was specified in the doctrine corresponding to finitely bicomplete categories (FM-categories) in which limits and colimits exist for all base graphs of finite size. We remain generally concerned with FM-sketches and FM-deduction within FM-Theories, but the prefix FM- will be usually be omitted. If *S* is an FM-sketch for a particular syntax, its FM-theory category `FM<S>` may be written simply *S*.

In Chapter 4, translation between theories was described by morphisms (translators) that respect the doctrinal structure – in other words as FM-functors. These translators represent systematic ways of converting between models, but because theories are *generated*, translators subsume an arbitrarily large amount of symbolic computation.

Morphisms between finite sketches are much simpler. In this context, finite sketch-morphisms will here be called codices in order to emphasize their role as illustrations or translations. A codex maps signature and constraints of a source sketch *S* coherently into signature and constraints of a target sketch *T*, and may be thought of as a vehicle for interpreting into concepts of *S* from those of *T*. Applying a codex converts any T-model to an S-model by simple operations of selecting and copying, without any deductive computation.

If *Z* is a category suitable for containing models, then any codex from a sketch *S* to *Z* is a model, regarded as an expression with structure *S* within the medium *Z* (in short, an S-expression in *Z*).

Equivalently, expressions are models of the theory $\underline{S}$ in Z (regarded as an FM-category), i.e. translators from $\underline{S}$ to Z.

6.1.2.2 Notated Language: a Category of Forms

Each FM-sketch S has an underlying graph, its signature, denoted |S|. Graphoid forms are models of these graphs, and thus are like abstract unconstrained expressions. Forms on a signature belong to a category $\underline{|S|}$ (a *topos*) that supports all the standard definitions of graph rewriting. The arrows between forms in $\underline{|S|}$ are transforms: natural transformations between models, as defined above.

The language defined by S (in Z) is then the category Mod Z(S), of S-expressions in Z, with transforms as arrows. This will be simply written $\underline{\underline{S}}$, when Z is understood from context. Although $\underline{\underline{S}}$ is a subcategory of $\underline{|S|}$, it need not share the same closure properties. The language $\underline{\underline{S}}$ may be deficient in transforms between its expressions and it may even be empty of expressions – in general there is no guarantee that the constraints of S are satisfiable, especially when Z is restricted to finite sets.

6.1.2.3 Media and Models

For most purposes we take the medium Z to be some category of finite sets, perhaps imagining sets in the given context, of geometric space, coloured pixel arrays, perceptual images, concepts and computer storage. If we restrict ourselves to formal reasoning, nothing is lost by regarding Z simply as an indefinitely rich theory category. If we wish to describe the nature of a specific medium more closely, this may be done by sketching a theory $\underline{M}$ of the medium, and factorising an expression in Z into a codex to $\underline{M}$, composed with some M-model in Z.

Let S be a syntactic sketch and $e: S \rightarrow Z$ be an expression.

Let $\underline{M}$ be a theory and $m: \underline{M} \rightarrow Z$ a model – i.e. a concrete medium.

Then e can be expressed in the medium m provided it can be factorized via m:

$$\exists p: S \rightarrow \underline{M} \bullet e = p;m$$

Expressions in Z are called concrete in contrast with abstract expressions in $\underline{M}$.

For the constructions demanded in Chapter 5, Z must have uniquely designated finite (co)limits, whose apex objects are referred to as *canonical* constructions. Thus, for example, *the* product of two sets in Z is a specific set that represents all the ordered pairs.

Since we rarely have cause to discriminate between isomorphic models, for computational purposes we could choose as medium a *skeleton* category *N* of natural numbers. This has a single representative for each size of set; each number is the set of its predecessors, and the arrows are all functions between the sets.

6.1.2.4 Meta-Schemas

In order to discuss the design of notation structure, we will need a new form of diagram to express relationships between sketches. These "meta-schemas" will be drawings of a more elaborate kind of sketch – one whose models are not in the medium of sets, but in a category of FM-sketches.

Sketches and codices are the 'objects' and 'arrows' of the category of Sketches, **Sk**. Since this category **Sk** is known to be bicomplete, it provides a suitable medium for mixed sketches; in other words we can use sketches with models in **Sk** to sketch the relationships between certain codices and sketches. This implies that a variant of the schema notation of the previous chapter can also serve to denote such a "meta-sketch".

The notation provided by meta-schemas could distinguish different kinds of sketch and codex in the richer structure of **Sk**. For instance **Sk** has a subcategory **Th** of theories and translators. Furthermore, a second level of arrows is needed to depict the transforms – arrows between parallel codices to a category[2] as in [fig 6.1].

The syntax of meta-schemas would therefore be based upon that of schemas, but extended to express transforms between codices and to distinguish different kinds of codices. In particular we may want to distinguish *deductive* codices that merely deduce extra structure, or to indicate the strength of the deduction – whether limits or colimits are involved. Details of such a notation will not be presented at this point, but informal graphical structure will be devised and explained when needed.

6.1.2.5 Notating Formal Proofs

In the formal design of a notation, there will be a practical need to establish logical dependencies between the facts formalized in a sketch. SIGN schemas can support structural reasoning by

---

[2] **Th** has the further structure of a 2-category, but **Sk** does not. Given two sketches S and T, we might consider what structure can be found in the set of codices (S → T), where the target T is not a category. Although transforms can be defined in this case, they do not form a category because the map formed by composing two compatible transforms may not itself be a transform.

depicting all the items and constraints involved in a chain of inference, but they are not designed to notate the deduction itself as a formal proof. Meta-schemas could be adapted to depict deductive stages between parts of a sketch in broader terms.

Though proof documentation is likely only to be of academic interest, the outcome of a proof is an important data-structure – a network of dependencies between sketch-items, useful in controlling computations on expressions. The dependencies could be diagrammed along with the schemas that visually motivate the deductive steps, but further special notation is not proposed here.

6.1.3 Specifying Semiotic Structure

We now turn our attention to considering what a full notation specification would look like. In Chapter 3 (§3.4.2) we observed that reported methods of specification take little account of semiotic properties such as layering. It is thus proposed to define the layered structure in a tectonic sketch that incorporates the syntactic sketch described in the previous chapter. By combining structure from all layers together with pragmatic items, this sketch could in principle specify some of the interaction between expressions and situations that was envisaged in (§4.5). We here examine its primary purpose, which is to guide interpretation and display of expressions.

6.1.3.1 An Outline of a Tectonic Sketch

To design a notation, the plan is to build a tectonic sketch that specifies all of its associated entities and maps, as are necessary for interpretation, display, translation or other processing. The flow of information, from graphical facts about an expression to properties of the referred-to situations, is to be carried in its network of maps and constraints and controlled by the dependencies between them. The tectonic sketch *K* for one notation might take the shape roughly illustrated in [fig 6.3]; the diagram shows a notation with four separated layers of syntax.

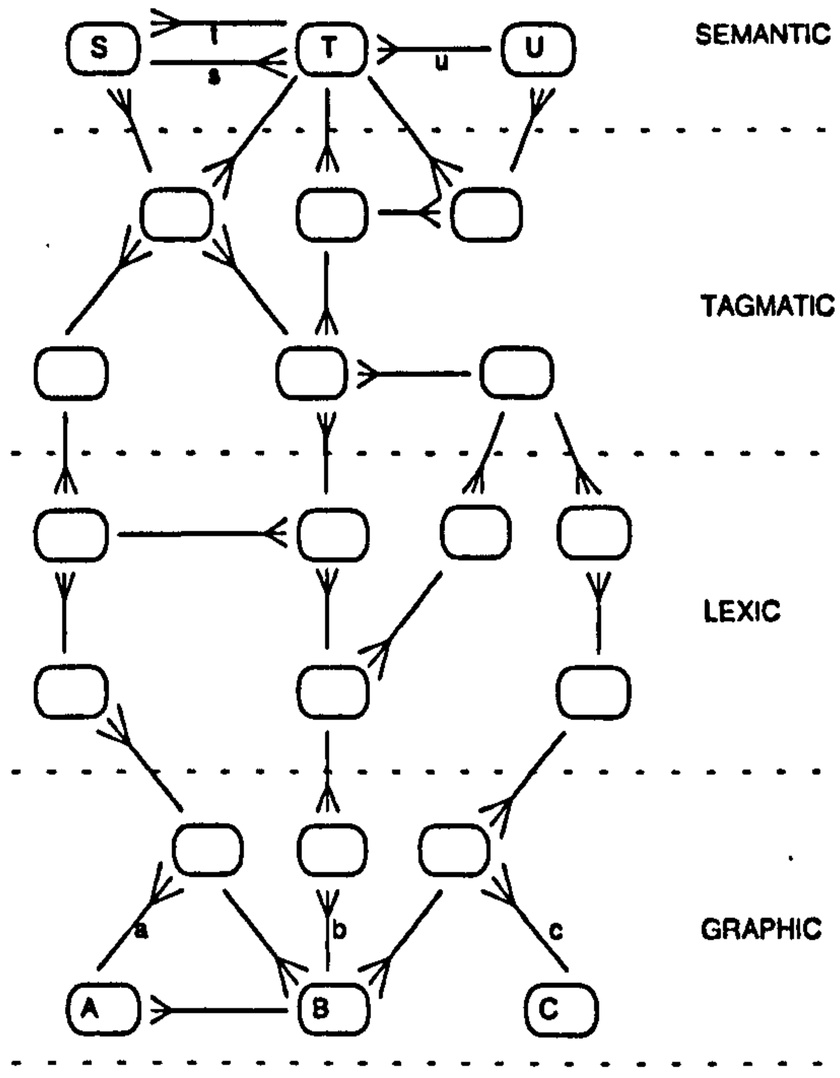


Figure 6.3 A rough picture of a tectonic sketch of syntactic structure.

This picture does not show any of the pragmatic constraints on how and when an expression may be used, necessary in a full semiotic sketch. Pragmatic items may connect at all levels – coordinating style and context, dealing with presuppositions, anaphora etc. These further contextual entities would supply referents for unbound names, and – in a programming context – might for example hold the structure of an automaton that executes expressed instructions.

The sketch could be extended to a community of notations with a common focus, so that different views of a complex system would be coordinated by the constraints. Within this global sketch would lie all the logical connexions. Its constraints would determine a dependency network for controlling information flow between representations and situations. This opens the possibility of constraint logic approaches for processing expressions – as suggested by some researchers in (§3.2).

6.1.3.2 Meta-Schemas for Tectonic Design

To facilitate the design, the tectonic sketch *K* would be built up from smaller modules, each defining a component of the formal structure. Meta-schemas provide a way to show the overall logical design, depicting the layers and deductive or analogical relations between them. This constitutes a second level of description, in which each sketch-name relates to a document of schemas that define it.

We can separate out the layers by means of four codices into *K*. Each codex embeds the sketch

for its layer in the abstract context *K*, as depicted in the meta-shema [fig 6.4].

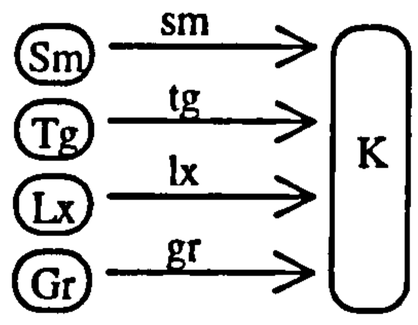


Figure 6.4 Layered sketches

Here the sketch *Gr* presents a limited theory of graphics that is appropriate to the notation concerned. This would be supported by a more complex general theory of the graphical medium, that is not properly part of the notational structure. Likewise *Sm* is only a small part of the semantic ramifications of the subject domain. This arrangement allows for overlap between the different layers, and also for some parts of *K* to be unrepresented in any layer. A K-expression `k: K → Z` contains items of all layers in *K*; it is correct in syntax, semantics and the rules that coordinate them, and is appropriate in context. It denotes a type of situation containing a drawing `gr;k` (a Gr-expression) with meaning `sm;k` (an Sm-expression). In this way the layers refer to different aspects of a situated expression.

If we were specifying a textual notation, the graphic layer might describe an expression as a two-dimensional array of characters; the lexic layer might define which sequences of characters are keywords, and classify other sequences as variable-names or numerals; the "tagmatic" layer might identify phrase-types and an underlying tree-structure, with cross-reference links between names; the semantic layer might determine the logical-types of subtrees.

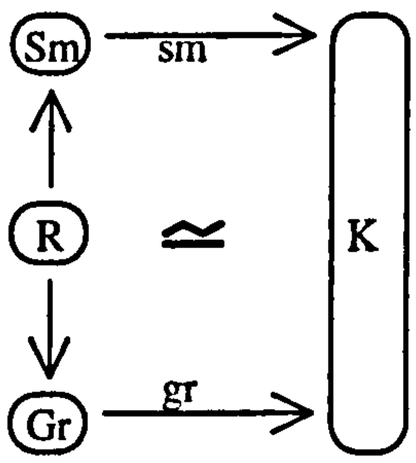


Figure 6.5 Direct graphical analogy

Further codices may be added to define analogies between layers, following (§4.5.2). For example, a direct analogy between some part of the semantic aspect and a part of the graphic would be described by a span – a pair of codices from a relator sketch R:

`Gr ← R → Sm`

In this case *R* indicates graphic items that are in 1-to1 correspondence with semantic items [fig 6.5]. The constraints of *R* will be satisfied correspondingly in *Sm* and *Gr.* More generally, a network of analogies of various strengths can be specified.

6.1.3.3 Relating Graphics to Semantics

The relation between graphic and semantic layers is not a balanced one. Design aims at an unambiguous interpretation for expressions, but conversely, the displayed expression need not be uniquely determined from semantics. It is usually preferable to have many ways of expressing the same statement.

In order to prevent ambiguity, it is necessary to ensure that the syntactic sketch expresses enough properties to determine semantic facts without the need for extraneous information. The strongest way to achieve this is to make the facts formally deducible. In such deductions, not all entities and maps in *K* are relevant; some take no part in interpretation, although their existence acts as a constraint upon the syntax.

> A simple example of this occurs when two sorts (*A* and *B*) of item are constrained so that there may be no fewer of sort *B* than sort *A*. To achieve this we can specify (in *K*) an injective map from *A* to *B*. Note, however, that this map describes only the operation of comparing sizes of the sets – it will not become part of the semantics *Sm*, nor need it be depicted graphically via *Gr*.

Thus in a suitably 'well-behaved' sketch *K*, internal logic would allow *exact* deduction (from a Gr-form) of all the maps in *K* that were salient to *Sm*. In such a case, each layer constrains all layers below it; hence not every Gr-expression would give rise to a well-formed Sm-form. Rather, every K-model would 'contain' a Gr-expression and its exact Sm-interpretation. Cognitive interpretation and production relies upon the deductive structure of *K* – thus the design must ensure enough logical dependency within *K* to support known operations and tasks.

6.1.3.4 Interpretation

Whereas the designer is concerned with deduction on sketches, the user has skills that operate constructively on models. In the case of an informal notation the interpretive procedure may be pictured roughly as follows.

> On being offered a visual expression, the reader favorably assumes that an interpretation of it exists. Where possible, she interprets new entities and maps from those present in the visible structure, by a variety of cognitive mechanisms. For those semantic maps that cannot be found by following deductive

semiotic rules, she has to derive their values by searching for a meaning consistent with background knowledge.

The purpose of formal syntax is to avoid the need for this kind of search by providing a deductive strategy for *exact* interpretation. Given a graphical expression, we can observe the values of sets *A*, *B*, *C*, and maps *a*, *b*, *c*, etc. of the graphic layer illustrated in [fig 6.3]. By obeying the constraints of *K*, it may be possible to calculate first the lexical sets and functions, then those of the syntactic layer, and finally the sets *S*, *T*, *U* and maps *s*, *t*, *u*, etc. of the semantics. Provided all the semantic entities and maps can be formally FM-deduced via intermediate structure, interpreting an expression becomes possible by direct calculation. If *K* does not support this direct unambiguous interpretation, it is necessary to search for sets and functions that resolve the constraints, given the graphical data.

Exact interpretation is to be ensured by relating *Sm* to some deductive extension of *Gr*. Models of *Gr* then have unique interpretations as models of *Sm*, as was established above. The procedure follows the codices:

$$Gr \leftarrow Gr0 \Rightarrow Gr1 \leftarrow Sm1 \rightarrow Sm$$

– where *Gr0* is a salient part of *Gr*, deductively extended to *GR1* (where '⇒' means a deductive codex); the codices from *Sm1* select the salient derived graphical aspects, and apply further constraints. In the same vein, the interpretive process passes through each intervening tectonic layer. For instance, this might be described by a sequence of codices:

$$Gr \leftarrow Gr0 \Rightarrow Gr1 \leftarrow Lx0 \rightarrow Lx \Rightarrow Lx1 \leftarrow Tg0 \rightarrow Tg \Rightarrow Tg1 \leftarrow Sm1 \rightarrow Sm$$

There are many variations on this theme; the important point is that such structuring can be described in some detail by a network of codices of different kinds.

The semantic sketch is likely to have constraints that are not a consequence of syntax, which can cause failure in interpreting some expressions. Such failure signals that the expression is syntactically correct, but inappropriate in the current (and perhaps any) circumstance. By tracing a sequence of deductions starting from facts of *Sm*, it may be possible to prove some properties that restrict lower layers. This will allow failure to be detected earlier in the interpretive process.

An example of interpretation is found in the task of assembling a syntactic sketch from a selection of SIGN schemas as mentioned in (§5.4.1).

What this description of the interpretive process implies is that a small number of simple cognitive steps separate graphic structure from semantics; moreover, these steps can be simply simulated in formal logic. The amount of effort needed for a construction in a model is apparently of the order of the size of the set of items involved – at least in a literal implementation in which each item occupies a storage location.[3] For this reason, the syntactic doctrine of the previous chapter excludes constructions which create entities for huge sets[4] on grounds that they might fail to be cognitively feasible. Were we to apply the suggested methods to the semantics and translation of expressions, a more powerful doctrine such as higher-order logic might be preferred.

6.1.3.5 Production

The reverse process, of graphically expressing a semantic idea, is not normally a direct calculation, but requires a search and selection of a pleasing layout. If it is hard to find any layout that works, the notation is impractical. To implement the passage from *Sm* to *Gr* efficiently, such searches could be guided by informal rules – for instance, extra constructions on *Sm* that specify favoured arrangements of items in lower layers. Further consideration of computational methods and their efficiency is outside our scope.

6.1.4 The Pictorial Design of Notation

So far this research has given very little attention to the pictorial nature of expressions, except rather abstractly in the notion of a graphic layer of syntax (§4.5.1). The discussion of this last aspect of design purposes to clarify the notion of a *drawn* symbol, and to observe what kind of formal descriptions are called for. How can we specify the appearance of a drawn expression?

Let us for a moment picture a notation as a 'semiotic house' bounded above and below by roof and foundation. The tectonic sketch then describes the interior, while the roof is the subject domain and the ground is the physical nature of the chosen graphical medium. These two boundaries touch the outer world, where description is independent of the notation.

We look briefly here at some questions about the lower boundary or 'ground' – regarding the

---

[3] – subject to a proper analysis of complexity, which has not been carried out in this work.

[4] In a higher-order doctrine, an *exponential* rule, for example, could construct the set of all subsets of a set, or the set of all functions between two sets.

designing of a pictorial realization for syntax. We ask how visual analogies can map explicit syntactic patterns into the implicit combinatoric geometry of the medium, as in spatial sequences and hierarchies. We seek adequate theories to support design of drawings, covering three different domains, two of which lie beyond semiotic bounds. For formally specifying the pictorial appearance of expressions we must know how drawings may be perceived and recognized. In order to produce or read drawings automatically, we need to know how they can be represented geometrically and computationally.

6.1.4.1 Embodying Syntax in Drawings

The contention here is that syntax is realized in an interaction between chosen properties of the pictorial medium plus further structural constraints applied to drawings. Descriptions of this interaction are essential for any claim that a drawing can become a precise portrayal. Ideally a designer should be able to prove absence of pictorial ambiguity, guaranteeing that viewers cannot be misled into deducing unintended properties – as in the failure of Euclidean geometric diagrams to support formal proofs.

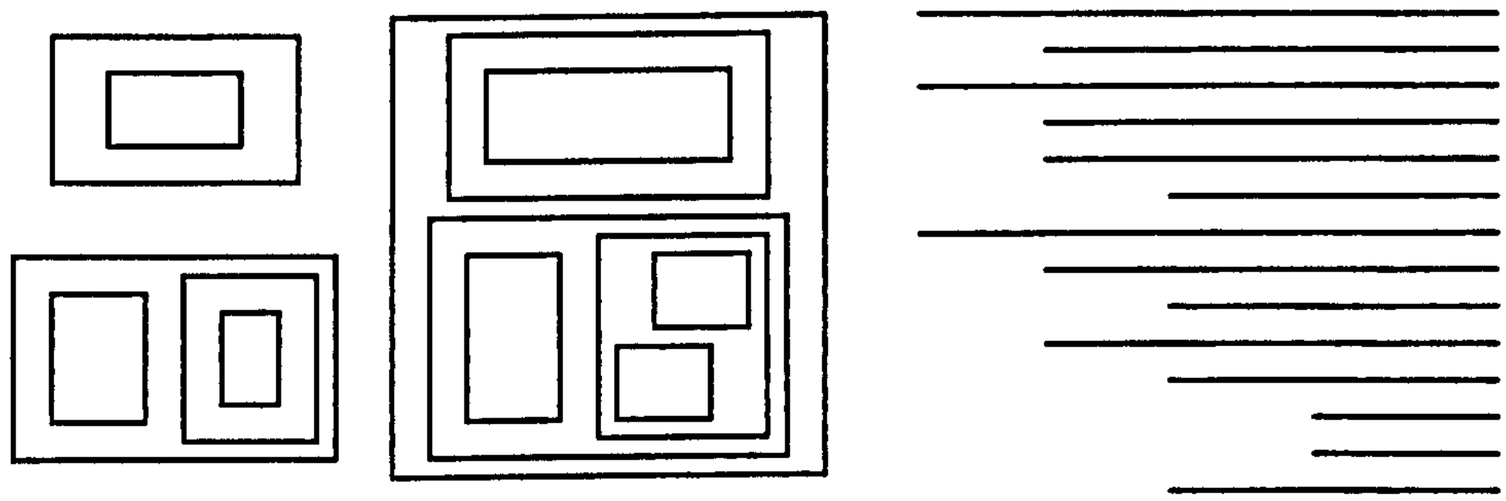


Figure 6.6 Two other ways of drawing a forest

There are many ways to endow a syntactic structure with a picture as body. Consider for example 'tree-like' structures (ignoring labelling of nodes) as in JSD. These can be drawn as node-and-branch arrangements such as [fig 5.24] in (§5.3.2), nested enclosures, or indented horizontal lines [fig 6.6]. Suppose for instance a diagram consists of a set of drawn rectangles that by convention may not overlap. The geometric relation of inclusion then enforces the syntactic relation of a branching hierarchy; thus the syntax is embodied by means of a conventional explicit constraint defined in the graphical layer of semiosis, together with constraints that are implicit in all drawings. The drawn form of the expression also satisfies many stylistic constraints that have no bearing on syntax. In the example, any non-intersecting closed curve would serve; rectangles are chosen since they are visually simple.

This kind of geometric analogy leads to economies in the notation, reducing the need for drawn linkages and expediting the learning of its significant features. In (§4.3) the structural analogy in the embodiment of semantics was referred to as a form of iconic mechanism. This is only the case when the salient graphical properties are easily recognized, owing to habits of perception or prior experience. The appropriate analogy may be indicated by a familiar cue – a visual metaphor.

6.1.4.2 Pictorial Theories

We saw in Chapter 4 (§4.3) that diagrams can call upon a wide range of principle of visual distinctions and oppositions in their expressive mechanisms. The design of these mechanisms requires an understanding of diagrams as *pictures*; for practical and computational purposes, it is necessary to formalize the pictorial properties. This study lies outside of the notational sign-system, but rather belongs to a general scientific perspective, whose laws are not restricted to simple logical doctrines. Inasmuch as diagrams are physical and cognitive objects, they can only approximately conform to any given theoretical design.

Some disagreement was noted in Chapter 3 (§3.2.2) about how much pictorial structure should be included in syntactic description. Several attempts at using spatial theories were found to suffer from a lack of grounding in both geometric and perceptual laws. Oliver Lemon (1996) reasoned for the requirement that a proper spatial theory should not admit models that cannot be embedded in a plane. Although this is a valid concern, it is not clear how careful we need to be in our axiomatization. Failure of planarity may be no worse than failure to fit the screen, but the logic would become over-complex if we insisted that only drawings within a given size and resolution be accepted.

Owing to the restrictions of the display medium (resolution and size of page), it is not easy to determine the class of syntactic-expressions which can in fact be drawn. Production of expressions entails a packing problem – a potentially complex search for a configuration that satisfies many geometric constraints. With theory we can reduce but not eliminate the need for trial and error.

Accordingly it is here proposed to divide the problem of pictorial theorization into several parts. Firstly there is the theory Gr that is specific to a notation – this is intended to rest upon the lower

boundary of the semiotic system. Below this lies a general theory of drawings as perceived objects. Then for processing we also need a theory of drawings as computational objects – and strictly, a theory of computer displays. None of these areas are well enough understood to yield an established standard theory, and they will not be examined here beyond the following brief comments.

6.1.4.3 The Graphical Theory

The lowest semiotic layer of a given notation is described by a sketch such as *Gr* in [fig 6.4], that specifies the most basic 'graphical' items and direct relations that make up expressions – e.g. the parts out of which lexemes are built. To fulfil this, *Gr* must encode relevant geometric properties, but without recourse to complex quantitative notions and topology. For example, a rectangular 'box' might be defined in terms of the incidences of its horizontal and vertical sides, and an arrow may consist of a constrained chain of straight segments joining head to tail.

Arbitrary models of *Gr* are abstract pictures in the style of notated expressions; they need not contain any significant patterns. Those models that are syntactically well-formed will be drawings of arrangements of lexical items. Different models of *Gr* that correspond to the same lexical combination should be recognizable as the 'same' expression.

6.1.4.4 Specifying Drawings as Perceived Objects

A model of *Gr* can hardly be called a drawing. In order for the design of a notation to take account of perceptual, cognitive and ergonomic criteria in its layout, we would need a psycho-physical theory of pictures in terms of universal items, valid for all notations. For a theory that can adequately explain visual analogy, we would need a general sketch of the pictorial medium, which could highlight analogies by matching of subsketches or derived subsketches. Building a theory of drawings is unfortunately a difficult problem.

An elementary theory of drawings might be devised by considering the normal capacities of human vision and perceptual *gestalt*-formation. This theory should be compatible with classical geometry, perhaps based on a qualitative approach, with concepts of association between elements derived from gestalt theories of perception (Palmer 1992). The appropriate description of the geometry should be determined from perceptual cognition and actions such as the control of focal attention. Viewers can follow the connectivity of lines and closure of curves, and establish

separation of regions – but only for a restricted range of shapes. They can match recognized shapes occurring at different locations, as with known words in a text – which is a good criterion to determine whether an item can function as a lexeme.

Perceived geometry is not simply Euclidean 2D, since the eye/brain responds strongly to depth and motion cues – as evidenced by many optical illusions (Kellman & Shipley 1991). Some notations exploit 3D notions that are constructed by the perceptual mechanisms; examples of this are the use of perspective transformations and the overlapping of items, with transparent or opaque features. Other salient geometrical features of shapes are: symmetry (see Appendix A), orientation, size and scale.

Geometry is only part of the problem. Although the shape of a character or other individual or discriminated mark may be definable via plane geometry, other attributes such as Colour, Texture, Pattern and Style take the variations outside of the domain of simple geometric theory. An understanding of the actions of drawing might also contribute to pictorial theory; the viewer of an expression may take account of *how* it was drawn, as well as what the result looks like.

6.1.4.5 Drawings as Computational Objects

For purposes of implementing graphical operations in a computer, we need to define an ideal data structure based on plane geometric concepts such as Points, Line-Segments and Regions. A geometric embedding of *Gr* must define its entities and maps as geometrical shapes, attributes of shapes and adjacency relations, etc. Standardization in computer graphics makes this a much more manageable task than the previous.

One way in which a computer-generated expression differs from a hand-drawing is that layout of elements can be modified by direct manipulation instead of re-drawing. If a viewer can re-format expressions, there may be the further benefit of reducing perceptual conflicts and ambiguities that might arise through poor layout.

Direct manipulation can be supported by a dynamic view of graphical geometry, following an analogy with mechanical linkage. A system of points and lines can be treated as a linked structure whose freedom of movement (and deformation) is restricted by arithmetical constraints. This is a practical solution which will be applied in the next chapter.

6.2 Computer-Aided Editing

In the light of the above methods, we now look in more detail at notational operations, taking the broad requirements of an editor system as motivation. The section discusses editing functions and the making of changes in theoretical terms. After exploring the properties of rewrite-rules on syntactic forms, the discussion points to the problems of applying rewriting techniques to editing, and suggests solutions. For several reasons, we discover a need for extra 'editorial' syntactic structure in order to make this feasible.

Although editing is not the only activity that calls for notation processing, it covers a wide range of operations of interest to us. Other important global operations such as translation and layout algorithms lie outside of the remit of this research.

6.2.1 The Demands of Editing

The practical tasks of editing have been mentioned in Chapter 3 (§3.3.1). The investigation here begins with an attempt to clarify the nature of the tasks and to find what processing is entailed. The analysis classifies the kinds of change that occur during editing of an expression and considers ways of managing both overt change and its hidden effects.

6.2.1.1 Making Changes

Editing involves changing the data of a displayed expression somehow; under this heading we can include tasks of creating, modifying, formatting and also translating expressions. Each task requires the control of changes to be carried out on expressions, whether directed by programs or by user actions. The conventional way of carrying out these tasks would be by drawing and re-drawing; in a computer-aided setting the system might automatically check or interpret the resulting drawings. With computer support much more is normally expected. An user can manipulate not just the graphical layer, but also the higher levels of structure which change the display less directly. In this case the computer takes on part of the task of producing the expression, as well as part of the interpreting. Editing is thus a collaboration between user and system, with the aim of regulating changes to information expressed.

Regulated changes can be divided into two kinds: those that can affect (formal) meaning, and those intended to preserve it. Creating and modifying cause changes of the first kind – they do not

necessarily maintain well-formedness at all levels. Translation and formatting changes are of the second – they affect lower layers of structure while leaving certain upper layers invariant.

- *Creating* an expression involves a supply of information, moving from an incomplete form to a fully well-formed form.
- *Modifying* a complete expression may require both removal and supply of information, or the joining of incomplete parts.
- *Formatting* involves the adjustment of graphical layout without affecting upper syntactic layers or formal meaning – either carried out for aesthetic purposes or to express further informal information, such as emphasis. Adjusting layout may also help a viewer to comprehend a displayed expression.
- *Translation* is an automated process in which an expression supplied in another notation is converted to the one being edited.

In each of these four tasks we would like to understand what the user is doing and how the computer system ideally responds.

In creating an expression, the focus is upon helping the user express ideas both formally and informally, by providing ways to add information. The user may be enabled to draw graphical marks directly, to select and place lexemes, or to choose syntactic arrangements ("tagmemes"). In each case the changes made are partially interpreted (by the system) into higher levels and also expressed in lower levels – and displayed on screen. A computer system would therefore need to store and coordinate data for all layers.

In other editing tasks, changes can be grouped according to the structural level that remains invariant – a flexible notation design will allow changes that are limited to a given layer and those below. Thus manipulation of graphical format will make no difference to lexical, syntactic and semantic data. Replacement of a lexeme by a synonym will not affect syntax and semantics. Rearranging a syntagmatic construction need not alter the meaning. These constrained editing options are supported by structuring the notation design into explicit layers.

Computed interpretation of a drawing takes place in (nominally) three steps. There is recognition of lexemes, parsing of syntagmatic patterns, and computing an abstract data-structure. Each step involves finding possible compound entities implied by the combination of items already found, and testing of the results against goal constraints. Where the test fails, 'backtracking' occurs – different groupings are tried until the goal can satisfied. If this search fails, the user may be

informed. A strategy for this search may be specified by a grammar. In designing a notation it is thus advantageous to rely more on constraining lower layers, by "moving" the semantic constraints down to the graphical layer, so that anomalous forms are rejected as early as possible in the interpretive process.

6.2.1.2 Methods of Managing Change

Making changes to an expression can therefore be a complex affair. We take it that a graphical editor system should help the user to control changes in content, form and layout of created expressions. In Chapter 3 (§3.3.1, 3.4.3) we noted two issues that are often raised in this respect: provision of syntax-guidance and direct manipulation of visible items. There we envisaged many ways of incorporating such facilities, ranging from direct manipulation of syntax (via a menu of transformation rules), to pen-based editing (e.g. free-drawing of changes over the displayed expression).

Free drawing would entail recognition as well as parsing computations. Operating on lexical and tagmatic patterns could avoid parsing problems and guarantee that applying rules would preserve syntactic correctness. It would, however, be dependent on establishing a complete and useful set of rules. We have seen that the strategy of specifying rewrite-rules to guarantee that syntax remain correct (if possible) is generally too restrictive.

Here we aim to negotiate a path between the extremes of freedom and restraint in editing. The model-theoretic method proposes to allow rewrite-rules to be flexibly devised and co-ordinated with the given syntactic constraints. These constraints can be seen as *goals* to be achieved – as in a text editor that accepts any entered text, relying on the user's knowledge of the language, while offering spelling and grammar checks at certain stages. The maintaining of syntactic correctness is then a matter of re-satisfying constraints that have been disturbed as a result of some local change made to the expression. A very lax approach to editing would permit the user to add, delete and rearrange items from any layer.

Immediate visual feedback is an important guide. Geometric properties may restrict change or at least act as a cue to the required constraints – such kinematic metaphors were remarked upon in Chapters 2 and 4 (§2.1.3, 4.4.2). Where syntax is not simply related to geometry, an offered repertoire of rewrite-rules may give welcome guidance for a less expert user.

6.2.1.3 Hidden Layers

Arguably, it is the lexical layer that often dominates the notator's attention during editing – in the same way that writers may concern themselves mostly with finding the right words. From the writer's perspective, syntagmatic and semantic data are, as it were, *hidden above* the visible drawing. On the other hand, the primitive graphical items on display are *hidden below* the lexical items recognized. Writers often do not consciously control the flow of meaning and syntax, nor are they usually paying attention to the detailed shape of words and type-faces used in printing. They are likely to be 'searching for a way of saying it' rather than searching for a meaning to express, or a grammatical way of generating an expression, or a geometric way to draw it.

Continuing this analogy, we see that when words are changed, the 'overhead' syntax and meaning as understood by the writer must follow suit; whilst the direct 'underlying' activity of shaping the new words is either habitual or mechanized. Typewriting and handwriting suggest two means of aiding the drawing of expressions. Either the editor system supplies ready-made lexical primitives, or it enables mark-making, in the case where the notator possesses specific drawing skills. If a lexical change is made, perhaps as permitted by an implicit rule, the other hidden layers of the expression are filled in by a transductive process – which will succeed when all constraints can be satisfied. In this way the user's attention is only diverted to another level when induced structure fails to meet some constraint.

6.2.1.4 Problems of Part-Formed Forms

During editing we thus expect some syntactic constraints to be broken. If the changes to an expression preserve full semantic well-formedness, the edited expression may successively approach the intended meaning. If they are only part-formed, the intermediate forms may acquire ill-defined meanings. Semantically, the goal becomes the achievement of a coherent meaning, being the one that was intended.[5]

Naive operations that simply delete or add a single lexemic or graphemic symbol are likely to break incidence properties as well – e.g. leaving 'dangling' arcs on a graph when a node is deleted. We should at least ensure that the changes preserve the more visible bonds. The

---

[5] In some cases, incoherence may be interpretable as less-definite meanings. This may provide an opportunity to develop an extended semantics that allows a certain vagueness and ambiguity.

propagation of a change may cause temporary breaking of hidden bonds, but restorative action should be automatic. An editor system must check the result of a change for conformance to (hidden) syntax, and report any failure to restore conformity.

6.2.1.5 Semantic Input

Where several notations are in use, we could allow for input in other modalities -- an expression in a different syntax or medium (e.g. gesture), or information from unexpressed context. The editor system would then translate the inserted input and incorporate it into the layout.

6.2.2 Editing in a Sketched Syntax

We now explore how the theoretical framework of tectonic sketches can explain processes of change that occur as a result of edit-actions. In considering how one might design a versatile editor, we examine goals of editing, maintenance of constraints on expressions, instantiation of sketches to make models, and the role of grammars.

6.2.2.1 Some Suggestions on Control of Editing

The above arguments suggest the following summarized account of the processes underlying an editing session.

(1) **Goals:** The procedure of editing is one of building and rebuilding a partial expression, with the goal of obtaining a model that satisfies all required constraints. The response of an editor to a local change prompted by the user causes new information to be propagated upwards and downwards through the layers, as specified in the network of maps and constraints. Where constraints are broken, all maps dependent upon them will become uncertain. One formal goal of the session is to repair all such 'damage'; there are also informal goals. The output from a completed editing session is a well-formed form: an expression well-defined in all tectonic layers.

(2) **Context:** If the session is conditioned by pragmatic structure, the resulting form may be further constrained by a specification of the context which will contain the expression. This may be either a type of situation or details of an actual instance of this type – such as an existing document.

(3) **Focus:** The editorial protocol should enable the user to initiate changes on different levels of

structure, although only the lowest graphical level is visible in a direct sense. It should mostly engage the user in emplacing and arranging lexical items. The focus for formatting is graphical.

(4) **Fixed substructure:** An instantiated area of the sketch may be "held constant" in order to restrict the editing session – protecting information encoded in certain layers. For example, formatting preserves data of upper tectonic layers; documentary context may be fixed, as in (2).

(5) **Translated Input:** To translate an input expression, the system must interpret it and re-produce it into the target-notation's form. The input need only be partially interpreted to some least abstract (semantic) level in order to make the conversion. In a known (instantiated) context, a similar operation may serve to give a change of view on a situation. In this case, the change may introduce extra contextual information.

(6) **Layout criteria:** When the computer system is required to produce expressions, it must apply chosen criteria for automatic layout. This can be user-assisted via the formatting protocol.

(7) **Prompting change:** Rewrite rules are one possible means of effecting a local change as directed by the user. The rules available to users have several purposes. Some are chosen to preserve certain constraints – 'strict' rules preserve more, 'lax' rules fewer. Others are designed to restore constraints that are commonly broken.

(8) **Editorial syntax:** Editing may necessitate the use of supplementary syntax – acting as a supportive frame for drawing incomplete expressions.

6.2.2.2 Maintaining Syntactic Conformity

The critical principle for control of editing is to decide which kinds of structural restriction are to be maintained during changes. We gain assistance in this from the items that make up a sketch, which fall into three different strengths. The strongest structural sketch-items are the maps which denote bonds of incidence. Next come the equalities that constrain the maps, and finally the cocones and cones that specify pieces and parts of expressions by their universal properties.

The proposal is to protect incidence bonding as defined in a signature, but pay less respect to the specified syntactic constraints. It appears that this policy can only apply to a chosen segment of

syntax that is the user's focus of manipulation. The rest of the syntax then acts as a body of hidden restrictions – which have the force of existential quantifiers. We are especially interested in the restrictive effect of layers that lie above the *editorial segment*, and how this effect is conveyed in the protocol. Lower layers only cause problems when layout algorithms are unsuccessful – when no acceptable drawing can be found.

Once the editorial segment has been chosen, it remains to determine which of its constraints will be respected by the changes that the notator may make – and by each rule governing these changes.

6.2.2.3 Full Expression

Given a tectonic sketch K, we see that the goal of an editing session is to create a full expression (K-model). The starting point will either be "from scratch" or there may be an existing fragment to be modified. Creating an expression is thus the process of making a model. In the general case, where the context is partly known, editing is a matter of extending an existing model that denotes the situations upon which the expression will impinge. A full expression is both interpreted and produced.

The notator accomplishes the making of this K-model by means of a directly manipulable editorial syntax that deductively supports the editorial segment (a codex into K). At any stage, the manipulated editorial form is a fragmentary expression that is produced and displayed, and may be extended by interpretation into a part-formed K-model. To produce an expression, the existing abstract editorial encoding of information is rendered into graphical form according to lower layers of syntax.

6.2.2.4 The Manufacture of Models

The problem to be addressed is that of model-making within the editorial syntax, which is assumed specified by a sketch *Ed*. Manufacture can be seen as a progressive *instantiation* of the full syntactic sketch *K*, with the intention to satisfy formal and informal goals. If *m* is the resulting Ed-expression, we require that it extends to some K-form *k* that is produced (made visible) and fully interpreted in *K*. Here we consider how *m* is created.

Instead of finding m as an object that satisfies *Ed*, we can extend *Ed* with structure that refines the definition until all its models are isomorphic to *m*.

A model (of *Ed* in *Z*) consists of an arranged set of items of various sorts, as defined in the entities of *Ed*. We can treat an item as a global element of its entity – a map from a canonical singleton entity *One* – which may be added to *Ed*. An item is arranged in an expression by selecting its relations to other items as defined by maps in *Ed*. These relations can be treated as equalities added to *Ed*, that involve the global elements.[6] An entity becomes fully instantiated when it is defined as a disjoint union of its global elements.

There are two ways of going about this instantiation. The first method is the familiar one of modifying expressions by adding and removing items.

The procedure can be illustrated by the simple case in which a notator creates a form by successively adding single items. The protocol for these additions can be ascertained from the signature of *Ed*. As the examples of Chapter 5 have shown, all items in an expression serve either as nodes or as polyadic 'links' – rather like verbs that relate subject and possibly several objects holding different roles.

In the sketch *Ed*, each outgoing map from a link-item is a 'limb' of the link, denoting a role that attaches to a specific sort of item. *Nodal* items are non-links, having no limbs; a *monadic* item is effectively a member of a set associated with its 'subject' item; a *dyadic* item is an arc that links a 'subject' item to an 'object' item of specified sorts; and so on.

Generally, no item may be placed in an expression until holders for each of its roles have been selected by the user. For example, on a directed graph, an arc may only be drawn if its source and target nodes are known. An isolated node may be added at any point – needing no role-holders. Thus an arc may not be drawn until a node is present.

Adding an item

The drawing of a graph may start by placing two nodes *p* and *q* (say).

– In the sketch [fig 6.7], this adds distinct maps `p,q: One → Node`.

To partly instantiate *Arc*, the notator draws an arc *r* from node *p* to node *q*.

– This adds a map `r: One → Arc`

with equalities `(r;source = p)` and `(r;target = q)`.

To complete this simple graph of two nodes and one arc, two constraints are added to the sketch. *Node* is forced to be the disjoint sum `(One + One)` with projections *p* and *q*, while *Arc* is forced to be isomorphic to *One* via map *r*.[7]

---

[6]This construction assumes that our notion of category is also defined upon Z -- its sets (of objects and arrows) are taken to be Z-objects.

[7]This defines the graph as a data-object, giving the cardinal number of items for each entity (two nodes and one arc) and all canonical projections.

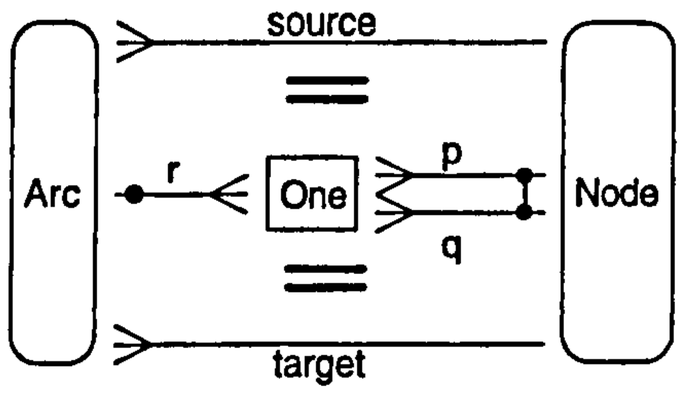


Figure 6.7

In general, the adding of items follows the hierarchy in the signature of *Ed* – a *pre-order* on entities – where entity *A* is "higher" than entity *B* if there exisits a path from *A* to *B*. If there are cycles within the signature, there will be different entities that are equivalent in this order, satisfying $A \leq B$ and $B \leq A$ (say). It follows that items of *A* and *B* must be introduced together. The problems of cycles are further discussed below.

6.2.2.5 Refinement by Grammars

In the second method of instantiation, *Ed* is regarded as a maximally vague definition for an intended Ed-model *m*. Hence we refine Ed = Ed(0) via a sequence of sketches Ed(i), by adding special maps and equalities that gradually define the intended model. The process ends with a sketch Ed(n) = M, that presents all the properties of a *particular* Ed-model *m* – from which the editor system must build a consistent full K-expression. The sketches Ed(i) are viewed as approximate (partly known) models, in which some entities and maps are not fully instantiated.

Since each Ed(i) is a sketch, it is not notated within *K*, and therefore the process requires extra notation to make the steps visible. This idea leads to the notion of an *editorial grammar* that generates Ed-expressions, using an extended syntax in which non-terminal items denote uninstantiated syntactic patterns. Forward rewrite-rules govern how each non-terminal may be replaced by a more specific pattern of items.

We can picture this editorial sequence as a river flowing from a single source, selecting one of many possible courses, resulting in a particular destination. As the river spreads, there is less certainty about the movement of a particular body of water within it. Whether by spreading and narrowing, or by selecting and meandering, it reaches its destination. Manufacture is not, however, a monotonic process; items and information can be removed as well as added. The river flows sometimes nearer to, sometimes further from the source, but always downhill, following its intentions.

6.2.2.6 Sketching a Generative Grammar

Following this train of thought, we consider how a sketched syntax can act as a grammar, to generate or parse an expression in some "visible segment" *V* of syntax. We observe two ways of sketching a grammar. The first is to specify its generated result – a well-formed derivation structure that incorporates a visible expression. The second way is to specify its parsing process – a whole search for derivations, including the set of successful ones.

The first way is compatible with that suggested in (§6.1.3). In this case a sketch *Dv* of a derivation corresponds to the upper layers, including *V*, of a tectonic syntax *K*. An ambiguous grammar would allow many derivations for a given well-formed V-expression. The second way is interesting because parsing is unique even in cases of ambiguity; it makes a sketch *Ps* for all (partial) interpretations of any part-formed V-form being edited. The advantage of this method is that the interpreting of a V-form is encoded in *Ps* as a generate-and-filter process which can accommodate ambiguity and failure. The disadvantage is that the model generated may diverge and generate an impractical mass of possibilities. In the general case, parsing may not terminate.

To obtain a result *Dv* from a parse-process *Ps*, it is necessary to instantiate *Ps* under the constraints that force all visible items to be generated by some single derivation of the grammar. An example of these methods, applied to a string grammar, is worked out in Appendix B.

By instantiating a sketch *Ps* in a top-down direction, it is possible to mimic the effect of simple rewritings in the grammar. Working from the bottom up, starting with *V* instantiated (a drawn V-form), the constraints of *Ps* propagate upwards to give all possible (partial) derivations. In principle, these methods could extend to attribute grammars that are capable of interpreting visible expressions.

6.2.2.7 Propagation of change

We conclude that changes to a model of *Ed* initiate a process of change within *K* – a flow of instantiation. Transductive rules can be used to spread the changes from the editorial segment *Eds* to other layers of *K*. The rules are also strategic, directed towards the goal of finding a full K-expression. They separate into two autonomous rule-sets – an upward set of interpretive rules, and a downward set of productive rules. Assuming unambiguous syntax, the interpretive rules amend a network of canonical extensions to *Eds*. The productive rules are responsible for making

the edited change visible.

The rules for propagating change are internal to the system and hidden from the user. They implement a form of constraint logic programming; executing the program instantiates a computer-assisted proof that a model exists to satisfy the current constraints. The methods of applying a grammar amount to ways of organizing such proofs, as proposed in Chapter 4 (§4.5.3). The whole system of editorial rules implements a kind of collaborative constraint logic engine.

6.2.3 Rewriting in a Sketched Syntax

The preceding argument indicates that control of incremental change to a model leads to the idea of rewriting. Since general rewriting systems can effect arbitrary computations – as was noted in (§3.1.3) – they offer a means of implementing sketch logic, and in principle the various processes described in (§6.2.1). Here the theory of such rule-systems is analysed, to investigate how rewriting techniques may be encompassed by the proposed logical framework and used to implement simple manipulation of forms.

We see that editing involves a dynamic of structure-breaking and structure-restoring, which must be managed somehow. The problem raised is that of organizing local changes to a structure. Being just a matter of general computation, we can apply standard techniques and theories. An approach such as conventional programming, however, does not promise to shed light on the procedures of editing that have been discussed. The object of using a rewriting notion is to get an abstract view that is better suited to these processes, while still forming the basis for an implementation.

6.2.3.1 Theoretical Rewriting Processes

Rewriting systems offer a means of describing systematic change within expressions as the parallel application of local replacement operators called rewrite-rules. We take a standard definition that is general enough for our purposes.

The general notion describes 'local' change to objects of some category F. Change is defined by a *span* $G \leftarrow D \rightarrow H$, where object G changes to H, whilst D indicates the context: the (main) part that remains constant. The two F-morphisms are injective. This change is controlled by rewrite-rules; each rule is a span that acts as a template for a local change. Applying a rule results in a

small change according to the template.

The following definition is specialized to notational forms, where the morphisms are transforms (§6.1.2):-

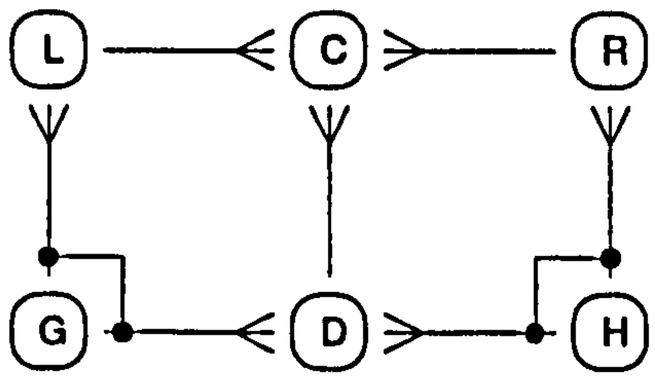


Figure 6.8 A double pushout

A **rewrite rule** takes the form of a span L ← C → R, where *L*, *C* and *R* are structural forms, and the arrows denote maps (transforms) that preserve the structure specified in some syntactic signature. The form *L* specifies a pattern which may be replaced wherever it occurs; *R* specifies its replacement; the interface *C* maintains attachments to the context. The patterns involved are concretely specified.

A (DPO) **rewriting operation** causes a change denoted by G ← D → H, where form *G* becomes form *H*, with a constant context of form *D*. This change is afforded by the above rule whenever the *double-pushout* diagram [fig 6.8] is satisfied; it states that *G* is formed by joining *L* to *D* at the interface *C*, while *H* is formed by joining *R* to *D* at *C*. Usually there is a further requirement that certain maps are injective – otherwise the rule may permit fusing and splitting of items.

Executing a rule on an expression involves several steps; the operation is a deletion coordinated with an addition. Firstly, a search is made in the expression for an instance of the pattern detailed in *L*, that is connected to its context as defined by the map *C* to *L*. Secondly, those items in the instance that are not maintained by *C* are deleted. Thirdly, a new pattern of shape *R* is attached via *C*, as detailed in the map *C* to *R*.

A change resulting from several successive local changes is also local. Thus we can compose changes, and thereby build a compound rule from the result of several successive applications of rules. Richard Banach's (1996) analysis of DPO rewriting on graphs, mentioned in (§3.1.3), could equally be applied to the graphoid forms assumed here:-

Rewriting system behaviour

Abstract rewritings take place on a *skeletal* category of graphs – i.e. one in which isomorphic graphs are considered as the same. Rewriting generates a category whose arrows are sequences of canonical rewritings; they allow a natural abstract explanation of a rewriting system's behaviour, via *event structures* with conflict (Winskel 1988). The construction of this category is somewhat awkward; in certain circumstances, automorphisms on graphs may force apparently independent rewrite-events to become causally dependent.

6.2.3.2 Sketches of Rewriting

Both the Span and DPO diagrams are in fact sketches, whose models are in the medium F that is some category of forms. In general:-

> A category $\underline{\underline{S}}$ of expressions defined as codices $(S \rightarrow Z)$ can – by cartesian closure of **Sk** – be represented as a sketch Z^S. Given a sketch *D*, its models in medium $\underline{\underline{S}}$ comprise a category of codices $D \rightarrow (Z^S)$ which is then isomorphic to the category of codices $D \times S \rightarrow Z$.
> In other words, D-models in $\underline{\underline{S}}$ are equivalent to (D × S)-expressions.

It is thus easy to encode both a change and a DPO rule as expressions in medium *Z*. If the category *F* of forms consists of models of a sketch *S* (S-expressions), then a local change is a model of (Span × S), and a rewriting is a model of (DPO × S). A calculation of products of sketches thus allows us to represent in diagrams the changes, rules and rewritings on a given syntax – in a natural way. Properties of changes and rewriting can be investigated by making deductions on these product sketches. (An approach similar to this is used in Appendix C).

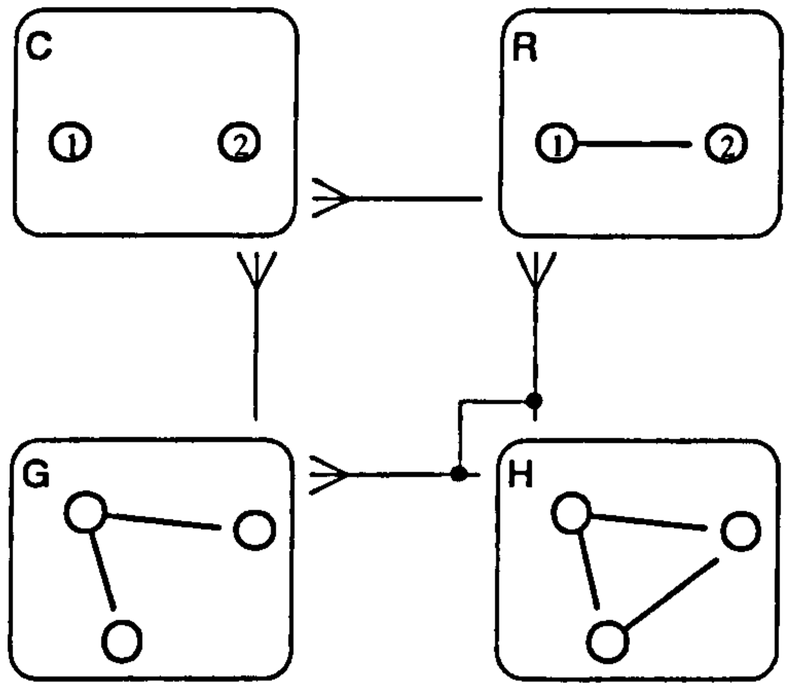


Figure 6.9 Adding an arc to a graph

6.2.3.3 Attaching an item

To illustrate, we look at the simple case of a rule that adds a single item to an expression – for example, adding an arc to a directed graph. Here the context is the whole graph *G*; the single pushout on the right is all that need be specified. The graph *R* will be the arc to be added, which must be accompanied by its source and target nodes. These two nodes are not added – they are the interface *C* which selects the location in *G* for the arc.

> In category-theoretic terms, the item (an arc) is being represented by a graph (R) and a morphism that selects the location of an arc in the result *H*. This is a case of the *Yoneda* construction; it allows any item of a form *H* to be represented by some morphism to *H* – which was part of the motivation for choosing graphoid structure for forms in (§4.4.3).

The diagram [fig 6.9] shows the operation. The nodes of *C* are labelled '1' and '2' to define the map (graph homomorphism) from *C* to *R*. The other maps are clear from the context.

6.2.3.4 The Problem of Cycles

Although we can always write rules for adding and deleting items, there is a difficulty with the DPO approach: there are many instances where adding a single item requires an infinite rule.

For example, let us take the case of a looped signature: `(f: N → N)` – there is just one entity N and one map *f*. The loop in this sketch leads to cycles in its models. A form on this signature might for instance be a cycle diagram *Q* of "nodes" (N-items), each mapped by f onto its clockwise neighbour. A form M representing a node is found (by the Yoneda construction) to be a semi-infinite chain of nodes; the morphism that selects a node in the cycle *Q* will, in effect, wrap the chain *M* around *Q* ad infinitum.

Even if we somehow accept such infinite rules, the simple operation of adding items does not suffice to generate all forms. When adding a new node to *Q*, it will attach outside the cycle. The pushout rule that adds a single node is of no use for enlarging a cycle. It seems we require an infinite set of rules, one for each size of cycle.

6.2.3.5 Transductive Rewriting

We wish to support the tasks of interpretation, production, translation, and the propagation of information which entail change from a form in one signature to a form in another. However, rules for changing forms by rewriting apply only within the *same* signature. Although in principle these facilities might also be implemented by means of a rewrite-system, its rules would manipulate a data structure more basic than that of the syntax – such as the labelled directed graph 'proto-notation' of (§5.4.1). In encoding rules as sketches and implementing sketches by rules, we fall prey to infinite regression. Accordingly, these questions will not be investigated here.

6.2.4 Editing by Rewriting

This final subsection synthesizes the ideas presented and explores a method of devising supportive structure for editing some target-notation.

6.2.4.1 Designing Rewrite-Rules for Editors

Although the behaviour of general rewriting systems is complex, we fortunately need only consider kinds of rule and rule-system that are appropriate to the tasks at hand. The preceding discussion in this section leads us to the following methods.

To apply rewriting to a notation editor, we must choose a workspace: a suitable category of rewritable finite forms – within the bounds of a logical specification of invariant properties and attainable goals. This must be a concrete category in which sufficient pushouts exist to apply the rules. The category need not be co-complete, nor have all pushouts, nor even all those that are mentioned in a rule-set. Because the rules refer to fragments of expressions, the workspace will contain forms that are part-formed from the viewpoint of the full notation syntax.

According to the previous reasoning, we need to select a syntactic focus for editing: a segment `Eds` → `K` of the full tectonic syntax *K*. Rewrite-rules are devised for a specially designed editorial syntax *Ed*, from which *Eds* is derived via codices `Ed` ← `Edd` ⇒ `Eds`. The sketch *Edd* excludes any *Ed* items that are not salient to the notation. Edited Ed-forms may introduce temporary items purely for construction purposes – visible place-holders waiting to be filled by more detailed information – in the same way as non-terminal symbols are treated in grammars. Editing has a goal of removing these items, which must be absent from a complete expression.

We may choose to edit by rewriting on |Ed|, the category of graphoid Ed-forms. This category has all the pushouts needed, but only protects incidence bonds and may be too lax to serve as a good workspace. In between |Ed| and its constrained subcategory Ed lie a range of partly-constrained categories. To design rules for editing, it is pertinent to discover what pushouts exist for these potential workspaces. Category Theory gives some general results on this – for instance, if we enforce only the equalities of the sketch Ed, all pushouts will still be found.

We want a generative rule-set to be *complete* with respect to a notation – capable of generating every Ed-expression; this can be achieved by means of add- and delete-rules for each sort of item. The rule-set need not be *sound*, in the sense that not all generated forms need be well-formed in *Ed* – and not all Ed-expressions need give rise to consistent K-expressions. Should we wish to preserve some stricter part of Ed-syntax during rewriting, we could seek rules in which a well-formed *G* would always give rise to a well-formed *H* (referring to [fig 6.8]). For instance, in a string grammar, string rewrites preserve the string-structure, even though the interface *C* for any

rule is not a string: it is typically a pair of nodes between which the substituted string sits.

6.2.4.2 Adding Branches to Trees

As a small example we can analyse rewriting on trees, following the definitions in (§5.3.2).

> Let the schema [fig 5.25] specify a sketch *FForest* for a forest of trees. The underlying directed graph of a forest will suffice as both editorial segment and syntax, with the sketch Graph defined as in schema [fig 5.2] . The embedding codex Graph → FForest then has two effects; it allows the set of trees to be constructed from the arrangement of nodes and arcs, and it also constrains the graphs to be well-formed trees.

Informally, it can be seen that if we start with an empty graph, two rules will generate all finite graphs:-

> R1: add an isolated node
>
> R2: link two nodes with an arc

We seek a set of rules that will only 'grow' forests. A third rule can provide a safe way of growing a branch on a tree:-

> R3: add to some node a "branch" consisting of an arc and a new target node

Now all three rules are pushout rules over the category of graphs, and R1 and R3 preserve the properties of forests.[8]

> Let *G* be a forest – a model of FForest. Then applying R1 to *G* will simply add another tree – or rather plant a seed for one. Clearly R2 may result in a graph that is not a forest, when applied to *G*, but R3 will always result in a forest.

The logic of adding branches to trees is worked out in Appendix C. We would also wish to prove that R1 and R3 can generate all finite forests from an empty forest.

Since the reverse of each rule serves to carry out the corresponding deletions, we can in this way provide a simple rule-set capable of editing forests, with a subset that maintains syntactic correctness.

6.2.4.3 Applying a Rewrite-Rule

When a rule is applied during editing, the search may be carried out by the user, with computer assistance in finding and indicating instances of the pattern *L* of [fig 6.8]. Since they are defined

---

[8]All the graphs in the pushouts are forests, but the graph-morphisms need not be forest morphisms. The latter must preserve trees and base nodes.

on the signature |Ed|, rewritings preserve editorial incidence, but may not always take note of Ed-constraints. Each rule may be designed to maintain certain constraints, while treating others as goals to be achieved. The constraints to be preserved should include those that are consonant with the pictorial realization, otherwise it may not always be possible to display the result in the notation's graphics.

A rewrite-application makes a change to certain functions denoted by maps in the syntactic sketch *K*. When these changes are propagated up through the layers of *K*, the system might warn the user of any constraint-breaking that results – indicating areas where further modification could achieve correct syntax. These procedures raise issues of editorial protocol which are not of concern here.

6.2.4.4 Removing Cycles

We saw above that problems in defining rewrite-rules can arise if the signature |Eds| has cycles, as is common. This circumstance is remedied by devising *Ed* so as to avoid the cycles, by relaxing certain incidence bonds into universal constraints. An example shows that cycles can be removed with little trouble.

We take the example of a looped sketch, that was considered in the preceding subsection.

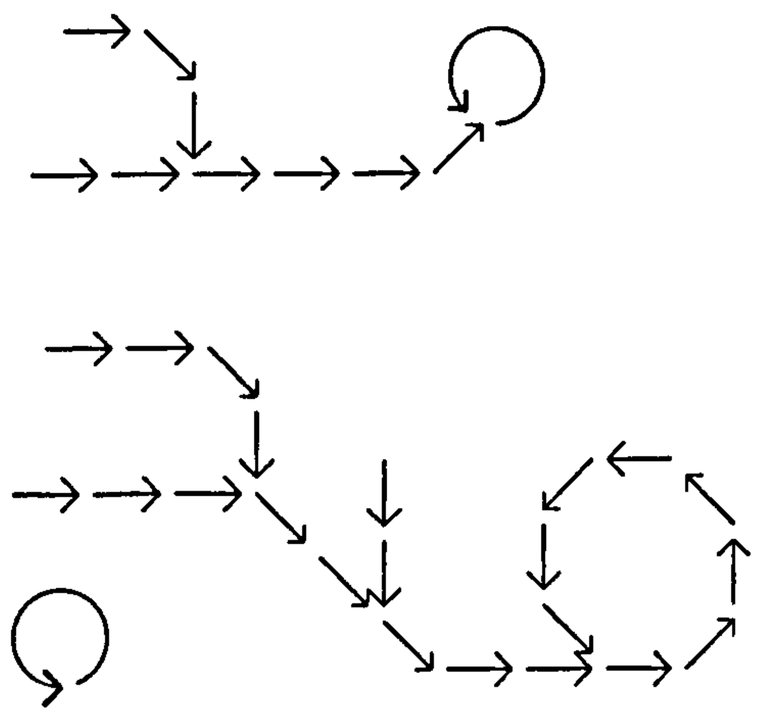


Figure 6.10 Function on states of a system

[Fig 6.10] shows a diagram *D*, whose arcs represent states of a system which is subject to a function. The diagram represents a set of states and their accessibility by the function; it also shows that there are two fixpoints where the state is unchanged by the function, and three subsets of states that indicate invariant properties under the function (Jay 1991). The sketch for this semantics is based upon a singe entity *State* with a single looping map `move: State → State`.

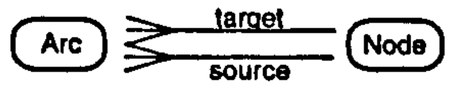
We can represent the semantics *Sm* by a schema:

which shows how the fixpoints and invariants are constructed.

The diagram [fig 6.10] suggests a syntax in which states are represented by the arrows of *D*. In order to manipulate these arrows individually, we need to find a diagram *Arr* that corresponds to an arrow, which we can map into *D*. We can then select any arrow with some morphism a: Arr → D. As was noted above, *Arr* is the semi-infinite diagram:-

→→→→→→→→→→→...

Inspecting *D*, we see that any arrow and its successors take the shape of a tail of length *m* followed by a cycle length *n*. In order to be able to delete any arrow in *D*, we are forced to have an infinite supply of diagrams Arr(m, n) to cope with every possible size of cycle and tail.

The solution is to choose as editorial syntax a directed graph whose nodes are the feet of the arcs. We take the schema for *Ed* to be 

as in [fig 5.2].

The schema [fig 6.11] relates *Ed* to *Sm*. The constraints specify an isomorphism between arcs and states, and state that the function *move* corresponds to the *target* map of the graph, via the *source* map. This source is constrained so that each node is a source, of at most one arc. This ensures that there is a bijection between arcs and nodes (in all models); *source* has an inverse that can be constructed by internal logic.

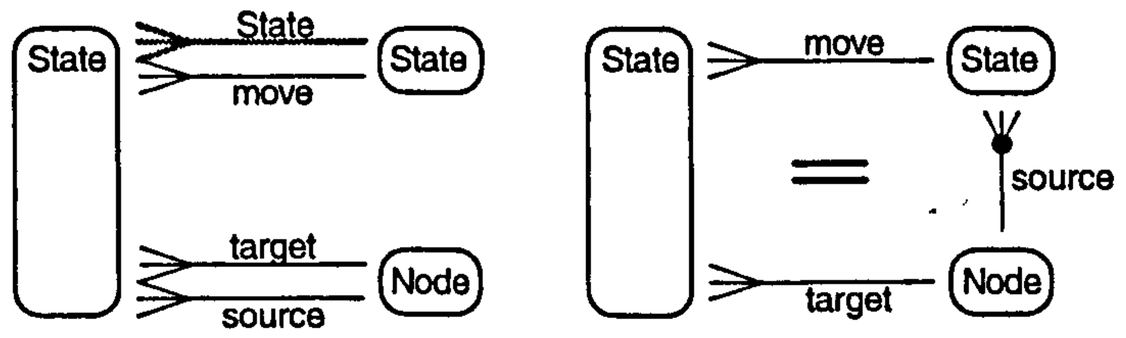


Figure 6.11 Semantic-editorial sketch for function diagrams

The isomorphism between *Arc* and *State* is easily maintained during editing, by permitting no arc to be added without its accompanying node. Only two rules are needed to allow any (Sm-)well-formed graph to be generated.

6.2.4.5 An Editorial Framework

It follows that support for editing requires the design of an hierarchical (acyclic) sketch – defining a more flexible, partial notation with a relaxed and extended syntax that is amenable to edit operations – whose expressions can be transduced into the target-notation. This editorial syntax should ensure that full expressions can be built up in a simple manner by means of rewritings and constraint goals.

6.3 Summaries

Here we recall the themes that have surfaced in this chapter, and end with an account of the approach to editing.

6.3.1 Themes and Topics

This chapter has explored various ways of developing the proposed tectonic theory, in support of notational design and computer-assisted editing. Three themes have been followed:-

- An encoding of semiological structure underlies all processing of expressions in a designed notation.
- The producing and the interpreting of expressions requires a "transductive" capability, to convert information between semantic and graphic forms.
- Mechanisms for reasoning about syntax also offer support for regulating change to expressions.

6.3.1.1 Deduction and Interpretation

Logical support needed for these processes has been supplied in terms of doctrines, sketches, theories and models. On this basis, notions of a *codex* between sketches, and a *transform* between expressions have been defined. Exact interpretation relates notation semantics to a deductive extension of its graphics. The effort needed to interpret an expression is related to the power of the logical doctrine defining the deduction.

- A mathematical "graph-based logic" has been applied that defines a notion of formal theory, independent of any particular presentation, in which deduction follows algebraic procedures.
- Deduction on sketches broadly simulates the 'cognitive effort' in an interpretive process.

6.3.1.2 Tectonic Design

A method of notation design has been outlined. This is achieved by building a *tectonic* sketch, whether for a single notation or for a community of notations with a common focus. A further meta-schematic notation helps depict deduction, analogy and embedding between segments and layers:-

- A tectonic sketch is a modular system of sketches defining a layered syntax for each notation, which may include semantic and pragmatic constraints.
- Meta-schemas are diagrams for the logical design of codices and sketches in tectonics.

6.3.1.3 Drawings and Pictures

The graphical layer of expression syntax represents only an abstract notion of a picture, that must be embodied in a physical medium. The chapter has noted a need for two general pictorial theories. One would provide a computational data structure based on plane geometric concepts, to support direct manipulation of a visual display. The other would be a psycho-physical theory of drawings, to provide a basis for defining spatial and other graphical analogies.

- The pictorial realm of a notation involves an interaction between chosen salient properties of drawings and added conventions of syntactic constraint.
- Spatial analogy leads to economies in the notation, with fewer conventions needed.
- The salience of graphical properties is affected by perceptual and conceptual habits, and is not purely a matter of geometry.
- A theory of drawings could test for pictorial ambiguity and solve "packing problems" in expression layout.

6.3.1.4 Editing

The support needed for editing tasks has been explored, with a view to letting the user decide the degree of freedom and restraint in an editing session. A theoretical view of the editing process, as model-making within a sketched editorial syntax, has been suggested. :-

- Editing involves changing the stored data that results in a displayed graphical form, including forms that only partially obey the intended syntax.
- Editing allows the supply and removal of items, with the goal of obtaining a well-formed form: one satisfying all the syntactic constraints required of an expression.
- Editing involves the breaking and restoring of structural constraints. Selected structural restrictions may be maintained during changes.
- Information of a change may propagate upwards and downwards through the syntactic layers, to update both interpretation and display.
- Formatting allows graphical adjustment in search of a desired layout.

Creating an expression can be viewed as the procedure of **instantiating** a sketch; editing modifies an instantiation. In this view, a sketch is itself regarded as an unformed pre-expression. During editing, the sketch is extended and refined by instantiation, adding constants to represent items, and equalities to bind them together.

6.3.1.5 Rewriting

Rewrite-rule systems have been examined as ways of controlling editing. Editorial rewriting requires to be carried out in a specialized syntax separate from that of the notation, so that rewriting can cover sufficiently general operations on the part-formed forms.

- Sketches allow us to diagram the changes, rules and rewritings on a given syntax.
- An editorial sketch can act as a grammar for generating and parsing, by specifying a generate-and-filter process – thereby accommodating ambiguity and failure.
- To edit by rewriting requires the design of an editorial syntax defined by an hierarchical sketch. Editorial syntax should be consonant with the notation's pictorial realization. The rule-set must be complete: able to generate all well-formed forms.

6.3.1.6 Propagation

Editorial actions initiate a flow of change within the whole instantiated tectonic sketch.

Rewriting has been examined as a way of implementing deduction in sketches. Interpretive and generative grammars are understood as providing proof-search strategies. In order to propagate editorial changes efficiently, incremental rewriting is suggested:-

- Rewriting can describe 'local' incremental change to objects of a category. A rewriting system's behaviour permits concurrent propagation of change.
- Propagative rules implement a kind of collaborative constraint logic engine. Rules attempt to re-satisfy constraints disturbed as a result of local change, by rewriting the basic data structure of syntactic sketches.

6.3.2 Summary of the Editorial Process

With this chapter we come to the end of the theoretical part of the thesis. The above discussion and examples of the second section indicate a certain method of supporting editing, which is summarized here in preparation for the next chapter, where development of an editor is discussed.

6.3.2.1 Editorial Rewriting

In the envisaged editing method, the editor system affords control by means of an expandable set of rewriting rules. These *editorial* rules govern direct changes to a particular segment of the syntactic structure that is 'graphically accessible' – its items are easy for an user to locate and manipulate. This segment is defined by an 'editorial syntax', acting as a supportive frame, that is tied into the tectonics at several levels and which extends the notation to permit part-formed forms

(pff).

These part-formed expressions are commonly focused on the lexical layer of notation syntax, but may extend to upper layers in a more grammatical or semantic approach, or to lower layers in a graphical approach. Rules that govern rewriting may be combined or reconstructed without affecting the notation. A single rule controls a very limited kind of change – it defines the exact pattern of items involved.

The editorial syntax is constructed to have an hierarchical signature, which is needed to enable rewriting. The frame belongs to the computational context; it makes building easier, but is removed when work reaches completion. It may involve extra graphical structure, since the frame may have its own ('non-terminal') drawn items.

Editing is carried out by rewriting expressions in the editorial syntax. Editorial rewrite-rules must include constructor and destructor rules that enable any expression to be generated and consumed. The starting point for generation must be some minimal form in the signature, typically with entities instantiated as empty or singleton sets. The constructor rules do not constitute a grammar for parsing an expression – the history of an editing process need carry no significance.

Although the editorial rules are simple, they can cover very general operations on the part-formed forms being modified, because their effects are extended by transduction into the whole of the notation's semiosis. This is done by employing autonomous rule-sets which propagate syntactic change to other parts of the tectonic structure. A rule can remove, for instance, a whole connected component of a graph – if the component is a defined syntactic construction.

Chapter 7

A System to Aid the Design of Notations and Editors

Abstract

Here we find a description of a software system to aid notation design, the plan of which follows the principles in the previous chapter. Details are given of the system's purpose, its component parts and how it operates. A basic prototype is developed and partly implemented within an object-oriented environment, in order to demonstrate how the theoretical framework and principles apply in practice. The methods of development and the actual process of implementation are explained, with indications of where this activity has added substance to the theory.

The planned system is proposed as a research tool to explore the design tasks of specifying syntax and generating syntax-guided editors. A full system would allow a practitioner to build and formally document the design of notations for a known task, addressing all aspects of structure and helping develop editors.

The prototype offers more limited functions, as a generic editor for a notation whose syntax is specified by sketches, and whose pictorial realization is defined using a geometric theory. It contains an editor for the syntactic schemas, which it can interpret as parts to be compiled into a sketch.

An editor for a designed notation is defined by an editorial sketch and a set of basic rewrite-rules calculated from this sketch, which may be augmented by further combined rules. Direct manipulation for formatting is derived from geometric definitions. The prototype system does not yet incorporate facilities for deductive reasoning about syntax and properties of rewrite-rules. The working implementation currently supports generic editing and editing of syntactic schemas.

Chapter7.

A System to Aid the Design of Notations and Editors

"Given the short horizons of research funding in the current climate, it is incumbent upon visual language researchers to demonstrate that their work is useful." (Wittenburg & Weitzmann 1996)

We have now reached a point where theory should give way to consideration of testing. How might the theoretical work developed here be applied in practice? To address the question, this chapter puts forward a plan for a software application (AGENDA) for computer-aided development of graphical notations.

The AGENDA system (A Generic Editor and Notation Design Assistant) is proposed as a research tool for exploring the practical utility of the syntactic specification techniques described in the previous two chapters. It offers help in designing and specifying any notation's syntax, and also in generating simple syntax-guided editors. With this plan in view, details are given of a more basic prototype that has been partially implemented for the purpose of demonstrating the principles. Implementation and development methods are explained, and a narrative describes the actual process of building the prototype during the course of the research. .To end with, we consider what has been discovered through this activity and what contribution it has made to the proposed theory.

7.1 Outline of the AGENDA System

We begin with an outline of the principles on which the system is established, and the functions to be provided. A short narrative suggests how the system would be used, and describes its components.

7.1.1 Principles and Functions

Here we look at what the system is for and what it can do to help build notations and editors.

7.1.1.1 Purpose

The AGENDA system is intended to help in designing or modifying a 'community' of notations in some common context. In essence, AGENDA is a generic expression editor with a meta-editor

that allows the user to modify both the syntax of each specified notation and also the editor's own rules. For developing each notation, the system will allow its structure to be sketched. Precise design of notational syntax is accomplished through using SIGN as a conceptual tool to help build a sketch.

Taking the sketch as a specification, an editor for the notation can be built, by defining a set of rules and protocols for guided editing. Since the focus of this research is not upon the complex matter of editor design, the plan concentrates on supporting basic flexibility, and not on elaborate interaction protocols.

Notation Design is a technical activity that requires specialist skill and knowledge of the structural properties of many existing notations. It may be compared to programming language design. Designing a new notation (referred to below as a target-notation) involves several tasks: specifying its syntax, defining the shape and behaviour of the pictorial components that make up graphical expressions, providing editing operations, and establishing semantic and pragmatic connexions to its discourse context.

In order to check that a sketch does define the intended syntax, assistance in thinking about structure will be necessary. The system must provide some support for reasoning, though not necessarily for documented formal theorem-proving.

7.1.1.2 Principles for a Notation Design System

The principles of the design are derived from the previous chapter. As indicated in (§6.1), specifying a notation principally involves the development of a syntactic sketch, which typically divides into tectonic layers. This specification is to be achieved by means of the graphical notation SIGN, as described in Chapter 5. Semantic processing for SIGN can then interpret a composed schema in the context of the growing sketch, which seeks to define a target-syntax. In practice, several variants of SIGN would be available for depicting the different levels of semiotic structure.

To give full support to a notation-user, the System should help build editors, with interpreters that can check semantic properties and convert expressions to computational data-structures or other forms of representation for further external processing. It should be able to express external data in target notations, and translate between these notations where appropriate. It should check compatibility of an expression with other expressions in the context, perhaps in other notations.

These demands for operation on expressions may be satisfied in the ways discussed in the previous chapter.

As well as the essential syntax for the target notation, an *editorial syntax* must be created, with the intention of ensuring that expressions can be built up in a simple manner by means of rewritings. This special weakened syntax normally has an hierarchical signature, and may introduce extra *non-terminal* symbols, as explained in (§6.2). The technique of designing the editorial syntax involves the removal of cycles by the method shown in (§6.2.4). This approach provides freedom in editing by relaxing syntactic constraints. Constraints that are goals during editing may for instance force the replacement of any non-terminal items that were inserted to aid construction.

In coordination with the syntactic sketches, the pictorial appearance of both the target notation and its editorial extension is settled by drafting geometric realizations for lower level entities and maps. Lastly an editor is assembled as a weak grammar composed of generalized graph-rewriting rules, with editing constrained or guided by the defined syntax.

7.1.1.3 Principles for a Generic Editor

The tasks and processes of editing were explored in (§6.2.1, 6.2.2). An editor lets the user build and modify expressions in a particular notation. A *generic* editor behaves as an editor for a chosen notation when it is supplied with the data which specifies syntax and editorial protocol for that notation.

Editing actions are governed by constraints in syntax, which are either seen as properties to be maintained or as goals to be achieved. The system can help in producing expressions in several ways:-

Guidance protocols

Available editing actions may be filtered in order to maintain certain constraints,
unsatisfied constraints may be highlighted on the display,
requesting an action may result in warnings about constraints that will be broken,
effected actions may result in notices about constraints that have been broken,
certain constraints may be checked only on request.

By applying rewrite rules on displayed forms, the user may generate any configuration of lexical items according to their own knowledge of the notation. Constraint-checking in software can signal any discovered irregularities to the user, who may use this feedback to help arrive at a well-formed

result. Layout is adjusted in a separate manner.

Many notations are flexible in allowing much variation in layout without affecting syntax. For adjusting layout, the system would afford basic direct manipulation of the notation's graphical items, while respecting the connectivity expressed by the signature and graphical details. Such an ability to maintain geometrical connections is a familiar attribute of constraint-based graphical systems.

Formatting is controlled by means of a defined protocol, which determines how the incidence constraints are applied. A lexical items can be a compound of simpler parts which are sometimes modified individually. When one part is handled, other connected parts must adapt to the change according to connectivity rules: a set of geometric constraints and protocols. The constraints assert properties which the token must possess before and after any change, while the protocol determines how changes to parameters are propagated.[1]

7.1.1.4 Functions to Aid Notation Design

Essential functions deal with the pictorial definition as well as syntax, reasoning of various kinds, and interpretation of sketches must be supported. The use of meta-schemas and methods for design of analogies (§6.1.3) is not described here.

Realizing a graphical expression as a picture can be seen as a task of translation from abstract syntactic representation to concrete display data. Shapes of lexical items are composed of graphical elements that may be constant, or variable in size (or orientation, etc.) according to context; hence the translation may not be simply determined by positioning of lexical items.

To allow expressions to be drawn, such items must be given distinctive features such as shape and position. As discussed in (§6.1.4), this is best done by embodying the syntax S in a general, near-universal pictorial theory of drawings. At some level of syntax the form is decomposed into pictorial elements, which are treated as data for display algorithms to process.

As described in (§6.1.4), the representation of pictorial data is non-semiotic – independent of the target notation. Calculations of geometric constraints can therefore be built into the System, and

---

[1] Some kinds of geometric constraint, such as collision avoidance or global symmetry, would require warnings or automatic layout heuristics after a formatting operation – these are not considered here.

will not have to be changed for each new notation under consideration. For diagrammatic notations, we would expect a graphical layer of target-notation syntax to specify the smallest drawn elements of the diagram.

In order to offer assistance in designing, the System should possess some theorem-proving capacity. It should be able to track logical dependencies between constraints and use pattern matching on the sketch to apply inferences, following the logic doctrine, as discussed in (§6.1.3). The network of dependencies within the syntactic sketch is important because it determines how the interpretation of an expression is calculated.

Technical work on syntactic logic needs to be supplemented with reasoning by cases: the testing of sample expressions against the proposed sketch. This necessitates a way of displaying samples – even before the pictorial appearance of items has been decided upon. For this purpose, the syntactic signature defines a 'default' appearance (§5.4.1) – equivalent to a directed graph labelled with names of entities and maps in the sketch – that can serve as an intermediate proto-notation, as illustrated in [fig 5.41].

By an interpretive process, the system can assemble a syntactic sketch from the SIGN schemas chosen to express it. Further interpretation of the information in the sketch would result in a dependency network linking the constraints of the sketch. This network would be used as a basis for processing during editing in the target-notation. Each sketch known to the System can be referred to by name in a meta-schema.

7.1.2 A Narrative of Facilities Proposed

By way of illustration, a short narrative on the proposed AGENDA system is given here. It includes an outline of the subsystems and their functions.

7.1.2.1 Using the System

On opening the System, the user is offered a choice of working *contexts*. Each context offers a choice of appropriate notations, and also documents that have been drawn up in that context. The user may open a document and edit an expression for insertion into it, by running an **editor** for one of its notations. The System is capable of building a community of notations, described in a specification document.

Changes of meaning expressed may be made by general editing, controlled in various ways, e.g.:-

free-drawing (making gestures that are tracked and recognized as drawing acts),
guided graphics (adding or deleting graphical elements),
lexic rewriting, (entering shaped items and characters as a whole),
tagmatic rewriting (rearranging structure),
translation from another notation (via a *translator*).

The meaning of the expression being edited can be held constant, during:-

formatting, in which only layout may be changed, or
rephrasing, by selecting permitted syntagmatic variations.

Edit changes may be guided by syntax and documentary context; they can be checked while an edit action is in progress or after completion. The completed expression stands as a message that can be sent to the document in order to effect changes to it.

If available, a (partial) **interpreter** may be run to calculate and check the semantic structure of the expression in the context of the document.

A **translator** runs a translation routine that partly interprets an expression, and then re-expresses it into another notation in the common context. The re-expression is constrained by the situation in which it is requested. In so doing it may express information implicit in the situation, for example by replacing a variable with its calculated value.

A **semantic analyser** assists the writing of interpreters which implement deduction or calculation by means of graph-rewrite systems. The analyser may also provide an interface with standard programming languages, for tasks that are fully computational and lie outside of the semiotic system of the target-notation.

From the editor the user may gain access to the **meta-editor** that enables changes to be made to the notation specification. In the meta-editor, the user can edit any schema belonging to the specification document. The schemas are drawn in standard variants of SIGN notation, suited to graphic, lexic, tagmatic, or semantic structural description. Editing is checked for conformance with the rest of the document. In addition to the normal editing functions, the meta-editor develops rewrite rules and constraint checks which make up an editor. It is accompanied by several assistants:-

A **sketch-compiler** is provided to process the semantics of the schemas. It compiles a new schema into the document, incorporating it in an interpreted sketch. It derives a network of dependencies in the sketch, and searches for internal conflicts and possible sources of inconsistency.

A **reasoner** is a 'deduction engine' that assists in logical construction and inference. The notation-designer can switch between editing a sample form and amending the schemas for its notation, using the reasoner to highlight consequences of a change.

A **drafter** enables shape and behaviour of graphical elements to be edited by direct manipulation, in conjunction with the information in graphic-schemas.

An **auto-formatter** helps design heuristics for automatic layout of computer-generated expressions, such as outputs from translation. These rules become part of an editor.

A **rule-maker** helps build rewrite-rules when designing the notation's editor, after the editorial syntax has been sketched.

There are also **libraries** of standard syntactic constructs and pictorial components such as boxes and arrows.

7.2 Developing a Prototype

This section describes work in progress on a prototype notation design tool, developed in order to make the proposed theory clear and concrete. Notes about the implementation give some of the details, and a narrative explains the structure of the software as it was constructed from the original ideas. The section ends with a summary.

7.2.1 Description of the Development

Here the development environment is described and the functions of the prototype are outlined.

7.2.1.1 An Object-Oriented Prototype

A prototype generic diagram editor is being built to explore and test the methods presented in the previous chapter. The software is written in the Smalltalk programming language and development environment[2]. Smalltalk is based on the object-oriented approach to programming, and supports the familiar 'desk-top' metaphors of interaction. It was one of the original languages to take this approach, and therefore has a relatively coherent structure in which every data item

[2]Smalltalk/V version 2.0 within Microsoft WINDOWS user interface, on an IBM PC (486).

(an integer, for example) is treated as an Object[3] that responds to Messages. The control structure of the language is imperative, though expressed by means of functional (lambda-) abstraction. Smalltalk was chosen in order to simplify the programming process and take advantage of a built-in library of graphical interface functions. Speed and compactness were not a prime objective of the exercise.

Smalltalk encourages development of software by building in new functions one by one on top of its own skeleton of graphical user-interface software. Development begins with a proposed Class hierarchy, which can be modified or enlarged later. Once the hierarchy is established, "Methods" (routines associated with a Class) can be added in any convenient order. These are compiled and incorporated automatically, without leaving the development environment.

7.2.1.2 Functions of the Prototype

The prototype system offers a more limited set of functions than the AGENDA proposal; for practical reasons, only a few essential operations are provided. It supports syntactic design using sketches and it has facilities for generic development of editors. The prototype includes an Schema Editor for SIGN, enabling the user to build and/or modify schemas, which contribute to a sketch for the target-notation. Pictorial design is carried out with a Shape Editor. A Sketch Compiler maintains consistency of a suite of schemas, which together can be interpreted as a partial or complete sketch of target-syntax.

A Meta-Editor assists the user in building an *editor*: a data-object that contains all the information needed to run an editor for expressions in the target-notation. Simple rewrite-rules defined on the sketch signature provide the main method of creating and editing expressions. A Rule-Maker helps develop the rewrite rules and constraint checks which make up an editor.

The prototype is limited to working with the design of a notation in isolation – not part of a community. Deduction and proofs of syntactic properties are not implemented. Semantic and pragmatic processing, such as interpretation and translation of target-notations, are also not implemented in the general case. For simplicity, graphical shapes are limited to line drawings plus ellipses and rectangles.

---

[3]This term and others used in object-oriented programming are capitalized in order that they may not be confused with other usage.

7.2.2 Implementation Details

Next some notes are given on how the prototype is programmed, with a definition of terms that the program uses. The implementation takes a naive approach, following as simply as possible the theory and principles; it is not intended to adapt or improve upon other implementations. The implemented facilities for direct manipulation of graphical shapes during formatting illustrate one method of manoeuvering the layout of expressions.

7.2.2.1 Data Structures for Expressions

The sketch of a notation's syntax provides a direct way to represent an expression as a set of stored *items*. These items are held in a single family of several named sets – the *sorts* which make up its syntax, with one sort corresponding to each entity in the sketch. Items may belong to any of the tectonic layers, e.g. lexical sorts such as boxes and arrows, or semantic sorts such as statements. The connectivity of the figure is stored locally, in the items themselves.

> Each stored item represents a syntactic token in the expression, and its incidence with other items is encoded as an array of *links* (i.e. pointers). There are various types of item, distinguished by their behaviour outside of the semiotic layers. Graphical items may be drawn (and displayed), and semantic items may be treated as data or operations by the system.

Expressions are stored for display as *figures*. A figure is a collection of syntactic and pictorial items arranged to satisfy constraints of syntax and geometry. The sets of primitive pictorial items that make up the displayed figure are constructed from shape definitions. The sketch for a notation defines its full syntax; the *forma* contains further data to define its pictorial form.

> The *embodiment* of a form is an interpretation of syntactic sorts and maps of the sketch S in terms of pictorial concepts such as geometric shape, graphical attributes, adjacency and enclosure. The syntactic sketch together with details of this pictorial realization is termed the *forma* for the target-notation. A figure is a model of the forma for S, i.e. an S-form with data for drawing it. An embodied expression is thus represented as a data-object formed of the syntactic sketch and the associated shape definitions.

Editing is carried out on a *figure* displayed in a screen window, with commands available from pull-down menus in the usual way. The constraints in the sketch give rise to simple routines that check an edited expression for well-formedness, using linear searches and tracking of links. When expressions are small in size, these operations do not require complex algorithms.

7.2.2.2 Glyphs

During editing, some drawn items can be directly manipulated on screen. These graphical or

lexical items, which have distinctive shapes, are referred to as *glyphs*[4]. The glyphs supply a pictorial grounding for the target notation.

> A glyph is a type of shape which can be recognized and discriminated from others. Its identity is invariant under specified transformations and spatial deformations. A *locatable* glyph is invariant under translation; sometimes rotation and enlargement are also allowed. A *fixed* glyph is in an absolute or reference position, e.g. the rectangular boundary of an expression. A glyph may be composed of parts which transform separately.

A glyph consists of a sequence of vertices (or *pins*) upon which stand primitive pictorial elements: lines, rectangles, ellipses and curves. The locations of these vertices define the actual stance in which the glyph is placed on the diagram, which may be adjusted during formatting. A glyph is constructed from certain vertices called *pegs*, which behave as controlling parameters for the shape.

The shape of a glyph is specified by primitive geometric relations between vertices, which constrain or restrain them. A *restraint* may for example force a vertex to remain on one side of a given line joining two other vertices. A *constraint* fixes the position of a vertex dependent upon other vertices.

A glyph contains a sequence of parts which are *drawings*, and a sequence of parts which are frame *shapes* that may be scaled or rotated, or are *sites* for linkage. Attachment of one glyph to another is achieved by restraining vertices of the one to lie in a site of the other. These sites may be circular, linear or polygonal regions, for instance.

The behaviour of a glyph is the protocol required for varying its shape or position during formatting. A peg may serve as a handle for adjusting the glyph. When a peg is moved, the new stance of a glyph is determined by calculating each vertex in sequence, provided each vertex depends only on its predecessors.

Here are some examples:-

> **Boxes**
>
> A box is characterized by the thickness, shape and symmetries of its quadrilateral boundary, and sometimes by the colour and texture of boundary and interior. The boundary normally consists of straight edges and may have rounded corners. A box may be resized along two axes, but not rotated. Resizing does not usually affect the rounding on corners.

---

[4](literally, carved items)

Arrows

Simple arrows consist of a straight shaft, with a head and tail on the ends, for instance, or a 'flight' in the middle. An arrow may be rotated as a whole. A flight may be moved along the shaft, but its orientation remains fixed relative to the shaft, and its size is also fixed. Compound arrows have polygonal shafts, in which each junction can be moved independently.

Curves

Curves, which show continuous variation in direction, present special difficulties in characterizing shape and style and finding a natural protocol for adjusting shape. They can be regarded as polygons with many sides, that are defined to appear smooth on screen. Curves may be constrained to be below a maximum curvature at all places, or to be convex, or closed. Any point of the curve may be adjusted, causing a local region of the curve to change within its constraints.

7.2.2.3 Shape and Behaviour Editing

To support formatting, a Shape Editor (drafter) is required for defining glyphs. The editor provides for drawing and designing their shape and appearance, and specifying their linkage properties during formatting. It allows the geometric incidence between glyphs to be realized in concordance with syntactic incidence. The Shape Editor should be supplied with a library of standard glyphs such as boxes and arrows.

When designing the graphical realization of an item, the user can place vertices, join them with drawn lines, and constrain them geometrically. Each kind of constraint and restraint is symbolized as a drawn connector between vertices. The editor allows drawing and specification of handles for format operations. The user can test that constraints operate as intended, by using the handles to re-format the glyph.

7.2.2.4 Formatting Protocol

A formatter allows the user to move and modify glyphs in a figure without affecting syntactic form. This is achieved by providing handles on each item, thereby affording variation of the item's parameters (position, size, etc.) when the user selects and drags the handle. Adjustment of layout is permitted so long as all restraints can remain satisfied.

The technique of *handling* allows the figure to be treated as a program for exhibiting itself as an interactive display. By using a selected *peg* as a handle, the position or stance of a glyph can be directly adjusted on screen. Moving a glyph-site constrains movement of those glyphs linked to it so that they remain correctly attached.

During formatting, any glyph can be selected for modification by pointing to its body. The selected ('live') token then displays handles (e.g. small black squares) on its parts. In any formatting operation, the coordinates of the handle position directly control the value of one or two parameters, which may vary within limits. The protocol maintains various principles:-

- A physical analogy is used to help the user understand the operation.
- The cursor is always free to move, but invalid positions are signalled.
- The values of changed parameters depend only upon the cursor position and the old values.
- To tell the user when parameter values are restricted, the path of the dragged handle is constrained.
- The body of a token is treated as a handle when moving it as a rigid whole.
- For visual feedback, a 'ghost' is displayed during drag operations, to help the user achieve accuracy.

The protocol can be expressed as a rule-set, in which each handle indicates a choice of rule and a site of application. The implementation manages the propagation of constraints and the computation of displayed shape.

7.2.3 Implementing the Prototype

Next we look at the implementation work, which involved developing a SIGN editor and a generic editor. A narrative of the process of creating the program is followed by a summary of the program structure.

7.2.3.1 Phases of Development

The prototype has been developed incrementally, by applying the theoretical framework to progressively more general tasks. The ordering of tasks is broadly as follows:

1. Groundwork for editing of schemas and generic editing:

    a) Developing a working editor for a simple version of SIGN schemas
    b) Generalizing this towards a generic editor, supplied with the formal syntax of SIGN
    c) Defining geometric primitives and using these to build lexical items for SIGN

2. Developing the Figure Editor

    a) Choosing a method for sequential encoding of figures for disk file storage
    b) Developing a shape editor for design of graphical realizations for lexical items
    c) Providing routines for checking syntactic constraints
    d) Providing routines for applying general rewrite-rules

3. Developing the Meta-Editor as an extension to the figure-editor:

a) Generalizing the figure-editor with methods for modifying menus

b) Developing a Compiler for interpreting a schema within a sketch

c) Developing a Rule Maker for building general rewrite-rules

The current version implements 1(a, b, c), 2(a, b) and 3(a).

7.2.3.2 Implementing the Schema Editor

Task 1(a) was the writing of an editor (*SchemaEditor*) to enable creation of schemas in SIGN. Because the syntax of SIGN has not yet been formally fixed, this task gave insight into the difficulties of editing graphical syntax. It was immediately necessary to fix on the shape of the lexical items and decide how they were to be linked geometrically. It proved to be a difficult task to reduce the geometry to a small number of primitive relationships. Resolving these geometrical issues was an important step towards the development of a formatter.

The notional syntax for SIGN was adapted from that implied in Chapter 5, to provide an editorial vehicle and to simplify the initial development. The syntactic items naturally follow a loose hierarchy (a preorder), the lowest items being *boxes* and the highest being the *angles* and *ties* which connect *arrows*. For simplicity, ties were only allowed to connect a pair of arrows. No extra graphical items were introduced into the syntax.

> The lexical items of a schema are named *boxes*, *arrows*, *labels*, *marks*, *angles*, *ties*, *equals*. Each of these has a shape, and it was convenient at first to make this shape a separate Class (subclasses of *Shape*). Each such class holds the Methods for drawing a shape and testing the position of the cursor relative to its body (for selection purposes). The items themselves were represented by subclasses of *Item*:- *Box*, *Arrow*, *Angle*, and so on. Each subclass holds methods for initializing the shape and forming syntactic links.

The Class *Schema* holds all the Methods for searching patterns in the syntax (as preconditions for rules), for adding items, deleting items (and rewriting generally) and for displaying a schema.

It became apparent that the main problems of editor-design lay in the protocols for simple rewriting of lexical items. Adding an item, for instance, involves using the cursor to select the size, shape and exact points of connexion of the new item. Other problems arose from the lack of a syntactic specification for SIGN:-

> In SIGN syntax, certain connexions are indicated by geometric relations: e.g. an equal-sign must be placed in a closed region bounded by a pair of arrow-paths which both originate from some box A and terminate at some other box B. Such a pair of "parallel" arrow-paths can be constructed syntactically, but the property that they form a simple region empty of other arrows is a geometric constraint. To

simplify the prototype, the syntax of regions was ignored, so that the attachment of an equality to its two paths was not checked. As a consequence, some of the syntactic relations stored in the syntactic representation were not forced properties of the schema displayed.

A computation of geometric facts is then needed during each editing action. This problem of specifying this calculation was deferred until the relationship between geometry and syntax could be investigated in detail.

A schema is a plane graph whose edges, nodes and regions may have markings; the edges attached to a node have a cyclic order which is featured in syntax.

The first draft of the editor was thus specifically programmed, with every operation programmed individually. This enabled the basic structure of the user-interface for the editor to be developed. In the process, a lot was learned about editorial protocol and pictorial definitions.

7.2.3.3 Abstracting the SIGN editor

Task 1(b): The next stage involved progressively modifying the program by generalizing all syntactic concepts, so that the editor would take on the appearance of being a generic diagram editor *specialized* to SIGN syntax. Methods for the generic *FigureEditor* could then be written. This process of generalizing or abstracting the program made the code more compact and flexible.

The first step was to modify class *Schema* so that it became a special case of *Figure* supplied with the details of an editorial sketch for SIGN. This was straightforward for the syntactic data, but issues of geometric (pictorial) representation had to be decided in order to store all the data of a figure. The pictorial data was also be needed to support the formatting operations on schemas.

The final step of abstraction required a standard technique for the user to apply rewrite rules, and a standard representation of these rules. Only addition and deletion of simple items was supported; even this raised the difficulty of setting a protocol to allow the user to position a new item in accordance with geometric constraints.

Task 1(c): Defining geometric primitives

Tackling the problem of building lexical items for SIGN gave rise to the notion of 'glyph' described above. The implementation copes with geometric constraint-solving in a simple manner; cyclic constraints are avoided by giving each sort of item a priority based on its hierarchy in the syntax. The geometric components of a glyph are universal pictorial items.

As just noted, difficulties arise in regard to geometric properties of regions, which have to be calculated when checking syntactic conformity of SIGN expressions. Although it was clear how to program this in the particular case, it is not obvious how to establish a set of primitive geometric constraints with any confidence that they would be universally adequate.

7.2.3.4 Developing the Figure Editor

Task 2(a): Choosing a method for textual encoding of figures for disk file storage

Each schema-figure was required to be stored on disk, most conveniently as a text file. This presented no problems.

Task 2(b) Developing a shape editor for design of graphical realizations for lexical items

Glyphs are supported by Methods for drawing shapes and testing when the cursor touches a shape (for selection purposes). A glyph is drawn by displaying all of its drawings; the positions of vertices are taken as arguments to primitive graphic functions which draw lines, boxes and curves. For speed of calculation, it is best to store the whole array of vertices which fix the stance, and not just the pegs which define it. The shape editor is treated as a diagram editor, by giving a graphical notation for the components of a glyph, namely vertices, pegs, sites, restraints and constraints.

7.2.4 Summary and Discussion

To finish, the content of the chapter is summarized, and we consider what has been learnt in the tasks and processes described.

7.2.4.1 Summary

This chapter has proposed a plan and prototype for a system (AGENDA) to aid notation design, based on the principles in the previous chapter. The AGENDA system would offer means of exploring and formally documenting the design and development of notations in a specified context, addressing all structural layers. The chapter does not describe a working system or a completed design for the AGENDA concept, and only a partial prototype is developed. The purpose of the prototype is to demonstrate practical application of the theory that has been presented in preceding chapters. Currently the prototype, implemented within an object-oriented (Smalltalk) environment, supports only generic editing and editing of syntactic schemas.

7.2.4.2 Processing and Graph Rewriting

The computational questions of how syntactic conformity is checked and how interpretation is carried out are not covered in this chapter. It is taken for granted in the previous chapter that algorithms exist for categorial construction of limits and colimits. General purpose algorithms can for example be found in (Rydeheard & Burstall 1988) in the medium of ML. Rydeheard & Burstall derive their algorithms from constructive proofs in category theory; in this way they show how to compute pushouts in general categories and also how to carry out term unification. The methods are easily applicable to graphoid rewriting, though they cannot provide an algorithm for graph unification. Since most of the constructions needed in our case are carried out on finite sets, there is no need for such a general method; we require only a simple algorithm for coequalizers.

In the prototype we have less need for these total algorithms, but more need for *incremental* algorithms that propagate syntactic constraints when a change is made. Grammars have a role in organizing this efficiently, which is why other work on grammar-based techniques is important.

It should be made clear that rewriting used to implement interpretive operations is not the same as editorial rules that are chosen in generating an expression, but the distinction is blurred when syntax-directed rules are used in editing.

7.2.4.3 Lessons Learned

The plan for the AGENDA system has provided a vehicle for addressing practical problems and discussing solutions. Since the practical exploration of the prototype development was interleaved with theoretical research, it was possible for practice to inform theory at several stages. Successful implementation gave confidence in applying the framework, whereas difficulties encountered were useful in clarifying and tackling a number of problems. For example, the problems encountered in pictorial definition and interpretive processes prompted the investigations recorded in the previous chapter.

The difficulties found in developing an editor for SIGN were caused not by inadequacies in the theory, but in aspects of the program that lie outside of the theory. Wherever aspects of syntax could be defined in a sketch, matters were easily resolved. Problems occurred in areas not effectively treated by sketches – such as the arithmetical operations of geometry. The attempt indicates many areas that require further research.

Chapter 8

Conclusions

Abstract

Here we find a full review and evaluation of the work in the thesis. We refer back to the literature review and the research objectives in order to assess the achievements in terms of the problems listed there. Most of these problems have been explored and the four objectives have largely been met.

A critique of the work discusses how it compares with other approaches and assesses what contribution each chapter makes to published researches in the same area. Problems that have not been treated or have not reached a satisfactory resolution are noted. The thesis is shown to offer attractive openings for further research into notation and analysis of sign-systems generally; this is emphasized by briefly exploring some possible future directions.

The evaluation points to the originality of the work in offering a new application of mathematics to questions of signification which have not in the past received much attention. The research expounds a precise diagrammatic method of specifying syntax, and charts the way to a formal semiotic theory of notation. Though much work remains to be done in this area, the theoretical framework developed in the thesis is seen to provide a firm basis for improved practical and computational support of notation in a technical context.

Chapter 8.

Conclusions

In this final chapter we review the work reported in the thesis, reflecting on the problems that have been addressed and evaluating the solutions proposed. At this point it is appropriate to ask how far the work has brought us towards an understanding of formal description and processing of notation, and to consider what further research is called for.

8.1 Summary of Research

The summary gives an overview of the research presented and lists the achievements.

8.1.1 Overview

First we review the thesis as a whole and then take each of its chapters individually.

8.1.1.1 The Thesis

Graphical notations were chosen as the topic for research because of their important role in technical work of many kinds. The focus of this thesis has been on notations in software development, which are often diagrammatic, but can also be as complex as the formal languages of mathematics and programming. Literature reviews have revealed that although a rigorous approach to notations is required, there is no standard formal technique for specifying their structure, and support has suffered from a related lack of adaptable tools for manipulating them. Research in this area is however increasing, and several diverse approaches are being tried.

In an attempt to encompass the diversity, this thesis has proposed a formal semiotic theory of 'notation tectonics', which treats notations as sign-systems and advocates a layered logical analysis. By applying the categorical Theory of Sketches as an uniform method, a schematic notation (SIGN) has been developed for specifying notation syntax formally – by means of diagrams – which can be extended to depict all aspects of semiotic structure. The method employs logical constraints, and supports graph rewriting as an operational technique. As well as providing a specification formalism, the mathematical 'sketches' may be compared and combined, making them suitable for a discussion of systematic metaphor and analogy in the design of a notation.

The thesis has described a new approach to building a tool for aiding the task of designing and

specifying notations. The AGENDA system allows notations to be specified and processed without recourse to a fixed graph grammar or spatial logic system. Such a tool promises a powerful method for the expert development of new graphical notations; it also offers the ability to help practitioners modify or adapt notations for specific circumstances – thereby granting them more control over their working aids. A prototype has been partly implemented to inform the research and test its practicality.

To a large extent, the research has thereby succeeded in its objectives. Many further avenues for investigation have also been opened up. Two of these are particularly important, as evidenced by other continuing work in the field: firstly the study of perceptual theories of graphics, which has a bearing on questions regarding ease of use, and secondly the study of deductive logic in relation to graph grammars, which pertains to the problem of grading the logical complexity of different descriptive formalisms.

8.1.1.2 Motivation and Methods

In Chapter 1 the reasons were given for researching the formalization of graphical notations, and a research method was proposed. The motivation rests upon the needs of software development practitioners to employ flexible notational techniques in all aspects of their work. Computer aids for notation processing are essential for a rigorous approach. The research objectives of formal specification for syntax and practical support for notation design were to be met by the methods of applied mathematics, starting from an informal discussion of the topic.

8.1.1.3 Reviewing the Area

Surveys and reviews gave attention to some of the practical and theoretical needs acknowledged in this area. In Chapter 2 a survey of the literature considered the few existing studies of notation, both early writings on mathematical logic and recent writings prompted by computer applications. A survey of writings on notations in software development revealed that they have become an important feature of methodology, but that no coherent body of theory exists to support graphical notation design. Practical problems with notation include difficulties for the learner, lack of explanation or justification of design, lack of formal definitions of structure and interpretation. The chapter suggested several avenues of research, the primary need being for mathematical and computational support.

in Chapter 3 an extensive review of related research considered formal descriptive techniques and tools to process notations. The review showed that there are no established techniques of syntactic description, though a disparate variety of approaches are currently being investigated. Many techniques follow methods used in linguistics, with various kinds of grammar; logical specification is also common, with some methods based on spatial relations.

The review of tools concentrated on editors and visual programming environments. In the past, tools to process notations have mostly been based on general programming methods, leading to inflexibility. Notations available in CASE tools are often closely bound to specific development methods. Although research into visual languages has begun to address the need for generic editors and universal syntactic formulations, users still have no easy way to influence notation design and usage. The chapter listed the many problems and avenues for solutions, and stated limited aims and objectives for the thesis.

8.1.1.4 Defining Problems and Solutions

Chapter 4 opened by establishing the boundaries of the research, examining the nature of notations and the roles that they play. An exploration of semiotic theory led to a discussion of how the meaning of symbols and structures is motivated and maintained, with a notation seen as embodying a stable relation between graphics and semantics. Signs in diagrams rely on metaphor and iconism in all levels structure, not just in the shape of lexical symbols. It was noted that motivation for signs, although important in justifying design decisions and giving help to learners, does not affect formal notation structure.

Semiosis was described as a (culturally local) network of temporary logical connexions amongst symbol configurations and their contexts, motivated by past patterns of experience and maintained by habit. Thus the logic is implemented through a variety of cognitive skills. No firm fundamental differences could be discerned between the various layers of structural description, though the relationship between higher layers becomes more complex. We observed that the structure of diagrammatic notation can be closely allied to the logical basis of its semantics, through graphical analogy. This reliance on analogy could render generative grammars inappropriate as a descriptive tool, despite their usefulness in implementation and for syntax of sequential text.

Based on research in the field of computational linguistics, an argument was made that tractable

logical systems are inherent in semiosis. The view was advanced that graph grammars are best treated as resource-sensitive logic systems, useful in constructing forms that satisfy specified constraints; they offer a model of limited-resource computation that is compatible with notational methods.

A theory of notation tectonics was outlined as a model-theoretic approach to an uniform foundation. Maps between formal theories were used to explicate the notions of analogy and translation. The aim of this very general theory – not based on an empirical study – was to provide a foundation for such a study of actual notations. Therefore important practical issues such as computational and cognitive complexity were only touched upon.

8.1.1.5 Sketches and the Schematic Notation

In Chapter 5 the logical basis for syntactic description was taken from Category Theory and the notion of a Sketch of a theory was explained. Through its analysis of logic, Category Theory appears to be the mathematical field best able to describe the workings of semiotic processes at a suitable level of abstraction. The structure of categorial logic lends itself to expression in a style similar to Entity-Relation Diagrams. This style was developed into a schematic notation for syntactic sketches, which was described and applied in a case study. The formalism puts the definition of structure on a firm mathematical footing, in which both notated expressions and syntax specifications are formal objects. Practical design of the schematic notation was discussed.

8.1.1.6 Questions of Notation Processing

In Chapter 6 the proposed methods were extended to the areas of semantics and graphics in a discussion of support for notation design and processing. Further theory covered processes of reasoning needed for design of syntactic layers and for interpretive and generative operations. Based on this, an outline was given of a method for building the tectonic sketch of a notation, and thus for expressing its syntactic and pragmatic design. The problem of embodying syntax in a pictorial medium was also considered.

The practice of editing expressions was analysed to discover what processes must be supported. The theory was shown to support generalized graph-rewriting for elementary editing operations, which are treated as steps in building a model of a syntactic theory. This investigation revealed

that a simplified or weakened adjunct syntax would often be needed in order to support flexible editing in this way.

8.1.1.7 A Notation Design System

Following the principles of this analysis, Chapter 7 presented a plan for a system to aid notation design, with the principal roles of specifying notation syntax and building notation editors. The object-oriented development of a simpler prototype system was described, followed by details of a partial implementation. Some problems were encountered in putting it into practice; formatting and layout problems were resolved by treating a diagram as a dynamic structure obeying a simple form of geometric constraint logic. The exercise was found useful in testing the basic notions of the theory and suggesting points for further research.

8.1.2 Achievements

In recording what has been achieved, we refer back to find which problems were successfully solved, and how this was accomplished.

By supplying a theoretical framework, the thesis offers a sound basis for practical and computational support of notation. The problems that were listed in (§3.4) have mostly been investigated:-

Difficulties in design methods D1-D4 have been addressed in Chapter 4.

Specification issues S1-S8 have been discussed and analysed in Chapters 4 to 6.

Problems of syntax formalisms F1-F5 and F8 have been addressed in Chapters 5 and 6.

Limitations of processing tools T2-T5 have been addressed in Chapter 6.

Difficulties of editors E1-E5 have been addressed in Chapter 6.

Chapter 6 has shown that formalization is helpful in ways H1-H4, H6, H8 and H9.

With regard to the objectives (§3.5.1), the following list indicates how they have been met.

1) The thesis provides an uniform framework for specifying notations, that:

   a) rests on established mathematics,

   b) supports reasoning about structure,

   c) does not rely on graph grammars and parsing operations.

2) The thesis provides a computational and mathematical foundation for design of notations.

   The method:

a) supports re-use and combination of specifications,

d) allows analogies to be described.

c) supports the use of rewrite-rules to operate on formal structure,

d) is compatible with constraint-logic approaches.

3) The schematic notation SIGN provides a clear diagrammatic way to communicate syntax.

a) SIGN has expressions similar in style to entity-relation diagrams;

b) it derives from diagrams used in category theory;

c) it enjoys full theoretical grounding in the framework.

4) The plan for developing a generic notation-processing tool is partly implemented with a prototype in Smalltalk. The development is made easier because the framework:

a) uses constructive logic that assists in implementation,

b) makes possible a detailed analysis of generic editing,

c) offers full formal support for the activity of notation design.

d) encourages a modular method of design.

8.2 Critique

Here we take a look at relationships between this research and other work; this is followed by a discussion of ways in which the work has failed to address or resolve important problems.

8.2.1 Comparison with Other Work

We wish to consider how the thesis improves upon, adds to, or contributes to other work on the topic. The following brief commentary recalls the approaches reviewed in Chapter 3 and considers how the methods proposed in the thesis relates to these other studies.

8.2.1.1 Other Approaches and Formalisms for Specifying Syntax

Many approaches are based on grammar techniques (§3.2.4, §3.3), of which the most general employ relations and constraints. Other researchers who start with a logical formulation are motivated by the computational convenience of unification as opposed to graph rewriting. In fact, declarative logic specifications can be regarded as a context-free grammars in view of the equivalence shown in (Corradini *et al.* 1991). Unification (Goguen 1988) is also important in equational logic, which supports executable algebraic specifications of syntax and semantics, via the linguistic notion of feature structures.

Some formalisms are intended to be purely descriptive, and are free to use logics that do not

guarantee a computable proof theory. In order to treat the graphical properties of notations, a few researchers explore spatial logic formulations; though those reviewed in (§3.2.2) are only applied to specific notations.

Amongst the grammar-based approaches reviewed, techniques adopt various **graph grammars** as a basis for defining structure; in order to overcome limitations, many go outside of the grammar paradigm, requiring constraint logic or programmed operations as well as rewriting.

Work on relational grammars (Wittenburg & Weitzmann 1996, Ferucci *et al.* 1996) is applied by some researchers. Müller & Lehrenfeldt (1994) use a version of Ferruci's context-free relational grammar, in which each rule rewrites a single symbol as a multiset of symbols, subject to a set of topological constraints between sequences of terminal symbols (which are graphical objects). Adjacency Grammars (Jorge & Glinert 1995) are related to Wittenburg's unification-based approach, with parsing controlled by associating constraints with productions; each adjacency constraint is associated with a function that finds the set of neighbours of a given lexical token.

Constraint Set Grammars (Helm & Marriott 1991) is an approach based upon Constraint Logic Programming. Constraint Multiset Grammars (Marriott & Meyer 1996) combine both relational grammar and constraint logic (§3.2.4), making it possible to define a generalized 'Chomsky hierarchy' of visual languages.

Some approaches use declarative logic as a basis for parsing, via **unification** algorithms (§3.2.1). These are able to cover a wider layer of syntactic structure by incorporating some spatial reasoning. Since specifications can be executed, the techniques are particularly suitable for defining visual programming languages.

The valuable work of (Haarslev 1995, 1996a) proposes a formal framework to unify grammars, semantic approaches and visual (spatial and temporal) reasoning. Description Logics – a declarative knowledge representation system based on inheritance networks and a term-rewriting language. The spatial logic describes qualitative relations between points, lines or convex regions.

Picture Logic (Meyer 1992) provides an executable declarative specification that is itself expressed in a visual language. A picture language consists of a set of spatial object types and a set of relation types. The work embeds Picture Logic in standard logic programming by implementing a new non-deterministic unification algorithm for picture terms. Though the result is

complex, less expressive Picture Grammars can be derived. Clauses in a specification are presented pictorially by borrowing shapes from the lexicon of the target notation.

Other **logic-based specification** methods and formalisms reported in (§3.2.1) rely on first-order logic and set theory. GDL (Welland *et al.* 1990) is an expressive textual language for specifying diagrams composed of typed nodes and links, by means of first-order logic constraints. PSN (Hekmatpour & Woodman 1987), developed for formal specification of graphical notation, admits first-order logic formulae, set-theoretic notation, function definition and a query notation for binary relations. VC^T (Serrano & Welland 1997) is a textual formalism for specifying syntax and semantics of diagrammed modelling techniques; it is based on set theory, and uses predicate logic to express semantic constraints. Z notation can also be used (David Gee 1995).

The generality of these techniques allows them to specify all layers of semiotic structure, which are often not clearly discriminated. Spatial logic is treated on its own in (Lemon 1996), which presents a complete axiomatization of 2D space in a modal logic of connected regions.

Algebraic and **semantics-based** approaches include VODL and VSDF (Üsküdarli & Dinesh 1995a, b) treat graphical and higher layers separately. VODL is a constraint-based declarative formalism for pictures; it describes the visual tokens and spatial relations that comprise lexical syntax. VSDF provide a visual formalism for specifying the syntax and semantics of visual languages. The syntax specifies a context-free term language, which is provided with an algebraic semantics.

Sorted logics are commonly applied in semantics. Region Connection Calculus (Gooday & Cohn 1996a) allows the specification, parsing and execution of expressions to be defined in a common language by means of the order-sorted logic LLAMA. A sorted logic (*inL*) is applied to semantic representation in (Klein 1987), which uses Kamp's Discourse Representation Theory: a version of first-order logic with a novel treatment of quantifiers, pronouns and anaphora. Order-sorted algebra is applied in (Wang & Zeevat 1996) to define a metaphor between expression and meaning as a partial mapping from a graphical signature to an application signature. A pictorial language would be characterized as a set of picture algebras. The nature of analogy is further explored in (Gurr 1996) in terms of homomorphisms between worlds of objects and relations.

8.2.1.2 The Value of this Thesis in Relation to Other Formalisms

The thesis brings together elements from many related studies, such as diagrammatic reasoning,

visual (programming) languages, mathematical logic, semiotics, computational linguistics, category theory, logic programming and graph rewriting. These threads are woven into a fabric whose strength is not dependent on any one particular study. This is an improvement on the many recent studies which are based on trials of some chosen technique; such approaches are restricted to a narrower focus, and are thus less credible as a candidates for an unified standard theory of graphical notation.

The approach adopted in this thesis is not based on coverage of a certain set of example notations, but justifies its choices with a theoretical analysis of semiotic structure. This policy for a general theory is less vulnerable to change when new examples arrive, and thus improves on those attempts that choose a specific definition of grammar in the hope that it will be sufficiently broad to cover the required range of cases of notation.

The research is not aimed specifically at extending any of the current approaches to supporting diagrammatic syntax and generic notation processing; nor is it intended to replace or to compete with work that investigates particular difficulties of syntax description and computational support. The research rather provides a framework which can encompass the approaches, making it easier to clarify their strengths and reveal the relationships between them. The thesis offers a mathematical method and a graphical notation which make this possible.

The relationship between different formalisms is, in general, a complicated matter that requires separate study. This work does not obviate the need to study spatial logic and the complexity of different grammars. The study of relative power of grammars in (Marriott & Meyer 1996) thus complements the thesis, which instead seeks to define a hierarchy in the complexity of constraints expressed in *sketches*.

The broad base of this research reveals that tractability is a central issue in notations, which is inherent in semiotic systems and is not just a consequence of a general concern for computational efficiency. Over-complex formalisms are thus inappropriate unless they offer a clear analysis of deductive effort. Over-simplified formalisms are only appropriate for kinds of notation that are known to fall within a limited range.

8.2.1.3 Using Sketches

How do the proposals of Chapter 5 compare with other logic-based formalisms? The idea of

'sketching syntax' is a radical departure from the predominantly grammar-based approaches currently being explored. The control of complexity in sketches is better understood than in grammars, because sketches benefit from a long history of research into logical systems, whereas graph rewriting systems are relatively new and difficult to analyse.

As a logical language, sketches are theoretically simpler and cleaner than those based on first-order logic and set theory with extensions. Sketches embody an elementary logical system and are also not complicated by any assumptions about graphical primitives (as is VODL) or syntactic structure (as are grammar approaches). In contrast to the approaches that are committed to a certain type of grammar, the approach in this thesis is to organize structure in a logical theory from the start, with syntactic sorts and logical constraints, and then to treat rewrite rules as a separate process of implementation.

Although the thesis does not demonstrate that all the various grammar and logic formalisms can be encoded in sketches, there is ample evidence that this should indeed be possible. For grammars, sketches can specify a derivation structure for the given grammar, as in the example of Appendix B. Implementing the logic entails a search for a grammatical derivation of an expression, which cannot in general be encoded in the sketches of Chapter 5, since the recursive operations required are only available in a stronger doctrine. The thesis does not explain how declarative logic can be expressed in sketches, though it is known that FL-sketches can encode Horn clauses. In (Goguen 1988) it is noted that cones of a sketch can be treated as generalized equations, embodying a set of constraints, for which a most-general-solution is simply a limit of the cone.

The use of a (subsumption) ordering on syntactic sorts or types is notable in several approaches, suggesting an indirect relationship with the feature-based grammars in computational linguistics (§4.4.1) which allow multiple inheritance. The definition (Dörre 1994) of a general feature structure is also similar to the notion of syntactic signature employed here, though syntactic sorts are not ordered in our case. Ordering of sorts would require an extension to the notion of sketch, but still lies within the general framework described in (§6.1.1), and should more naturally accommodate the linguistic aspects of notations.

Sketched syntax also rests upon an encoding of graphical properties. Spatial logics are not

addressed in the thesis, though they are relevant to the graphical and pictorial theories requested in (§6.1.4). The separation of abstract syntactic structure from pictorial properties helps keep the proposed formalism simple and general.

8.2.1.4 The Schematic Notation

How does SIGN compare with other specification notations? The other methods reported have mostly given little attention to how specifications are presented and made available to users. Previous notations for describing diagram syntax are either informal or choose a technical textual coding. Those whose aim is to express syntax pictorially (§3.2.1), borrow most of their graphical aspects from shapes in the notation being specified. Specifications in SIGN are made more accessible by taking a schematic form; the schemas are rigorous, being both formal and graphically independent of the target notation.

8.2.1.5 Other Methods of Formalizing Semiotics

The methods that the thesis offers can be applied far more widely than the special topic of notation, though they do not directly address the general problem of how to formalize semiotic theory. Although there is increasing interest in semiotics as a field for computational research, with applications in a number of disciplines, the only other comparable formalization available is that very recently developed by Goguen (1997).

There are many similarities between Goguen's mathematical approach and that presented in Chapters 5 and 6 here. Goguen espouses Category Theory; he works with sorted logic, and develops a notion of hidden algebras (Goguen & Malcolm 1996), in which hidden sorts fulfil a role resembling that of hidden parts in (tectonic) syntactic signatures of Chapter 6. His work differs in that he pursues an algebraic approach, so that specifications are (for the most part) executable by means of term-rewriting in OBJ. The language OBJ also places an ordering on sorts. Goguen & Malcolm's *hidden agenda* (*op. cit.*) , concerns one theme that has arisen in this thesis – that of combining different paradigms of logic implementation; it is especially motivated by the object-oriented paradigm, but it does not include graph rewriting.

It has been argued in this thesis that specification of notations should not be tied too closely to implementations, since this must lead to greater complexity in the descriptions. Sketched syntax can separate specification from implementation because the model-theoretic approach

distinguishes between internally provable and externally observed properties of expressions. This is more intuitive than standard algebraic approaches, in which the notion of an expression is identified with the class of all equivalent ways of generating it.

8.2.1.6 Other Work on Notation Design and Generic Editing

None of the works reviewed in Chapter 3 consider the task of designing notations in any depth, though the generic editors of (§3.3.2) have limited uses for this purpose. How can we assess claims that it is easy to build an editor with such systems? The AGENDA concept as proposed in Chapter 7 goes beyond the notion of a generic editor for diagrams, which assists design only in certain limited syntactic and graphical features.

As noted in (§3.4.3), researchers who have built generic notation editors have been generally concerned with gaining practical success on supporting a range of trial notations. They have not established an adequately broad definition of what constitutes a notation or an editing operation, a problem which this thesis addresses.

Grammar-based editing rests upon a set of rewrite-rules, whereas this thesis gives rewriting a different status:

> In the graph grammar approaches (§3.3, 4.4.2), expression structure is defined as a derivation, via rewritings that follow some fixed rule-set. These fixed rules do not on their own constitute editing operations. In contrast, in editorial rewriting proposed in (§6.2), the history of deriving an expression can be discarded – since interpretation is carried out only on the result, effectively by a parsing operation. Information on how an expression is drawn may give help in automatic interpretation. If the editorial syntax adds non-terminal symbols that denote 'hidden' syntactic items, information in rewritings need not be discarded because it reduces the effort of parsing.

The experience with graph-grammar-based systems such as DiaGen (reported in §3.3.2) has shown the need to define transformations on the derivation, which are propagated to the visible structure. This is in accord with the idea that flexibility in an editor can be achieved only by means of manipulation in an unrevealed accompanying structure. For Diagen, this was discovered by testing a prototype, and not through prior theoretical analysis of editing processes. The use of a syntactic specification in AGENDA avoids the need for a fixed type of graph grammar.

The small amount of system development outlined in Chapter 7 cannot, of course, compete in practical terms with large projects such as PROGRES (Schürr et al. 1995). The AGENDA system

would not go as far as the PROGRES system in offering complete notational support integrated with software development. Methods employed in PROGRES follow a layered approach to graphical parsing (§3.3.3), which is in sympathy with the reasons for layering advocated here. Whereas specifying notations in PROGRES requires the engineering or programming of graph grammars, AGENDA would see the task as one of building up a specification from re-usable parts, supported by reasoning.

AGENDA is not aimed at the design of user interfaces to applications, as in Escalante (McWhirter 1995). In principle, AGENDA is closer to a constraint-logic system, though not explicitly implemented as such. GenEd (Haarslev & Wessel 1996) and VisualGen (Chok & Marriott 1995) are similar in philosophy, though not in methods; GenEd aims to support reasoning; VisualGen allows flexible hand-drawn editing.

8.2.2 Unresolved Problems

The next task is to provide a critical summary of the work, assessing what has been overlooked, what remains unresolved, and how the solutions could be improved.

Is this a suitable topic for a PhD thesis? The amount of published interest in diagrams and visual languages has increased considerably during the course of the research, showing that the topic lies within a broad problem area that is perceived as important. The specific topic of notation in software development is less usual, and more manageable, but still raises difficult problems.

8.2.2.1 Survey and Review

Much of the literature in the survey (Chapter 2) is written from a software developer's perspective and tends to be prescriptive in approach. If the purpose is to ascertain how notations are used and what is needed to support them, we would do better to consult studies of the tasks of software design process for a more objective viewpoint. Ideally we would need models of the activities in software development in order to explain the problems with notations, but this is another difficult area worthy of further research.

There may be a bias towards an overly formal approach in the quotations chosen; design theory suggests that informal notations are at least as important in practice. It is hard to establish what are the needs for formal graphical notation design, since this has up to now not been a feasible

task.

The review of techniques and tools covers sufficient examples of the relevant approaches and difficulties. As pointed out in (§3.4.3), a more comprehensive review would look at CASE tools or data visualization applications, which could give more definitive information on how notations are used.

8.2.2.2 Balance of the Approach

Is there too much theory? There is a danger in developing a theoretical approach without backing it up with concrete examination of an extensive body of data. Despite the arguments made in Chapter 4, it may turn out that important difficulties have been missed. The value of the work lies in the conceptual framework that it provides, which can now be used for the detailed analysis of particular notations.

The work deliberately avoids the known practical problems of finding effective and efficient grammars, upon which other researchers have concentrated. It also avoids the difficult questions of pictorial perception and inference. The strength of the approach would be improved if these other concerns could be shown to be addressable within the framework presented. It would then be possible to evaluate and compare the varied methods reviewed in Chapter 3.

8.2.2.3 Tectonic Notation Theory

To validate the notion of a layered structure in notation, we require further evidence from cognitive analyses of perception and interpretation of diagrams or formulae. Study of evolutionary mechanisms could help explain how such cognitive abilities arose. Otherwise we should regard 'tectonics' simply as a convenient modular method for developing a formalized notation.

8.2.2.4 Choice of Mathematical Method

Is Category Theory appropriate? Despite being an established tool in theoretical computer science, Category Theory is unfortunately a difficult theory for the layperson to apply. It remains to be seen whether its mathematical intuitions can be conveyed well enough for the practical purposes intended here. There is no obvious alternative that would be as versatile.

The level of category theory applied is purposely naïve and elementary. More sophisticated categorial methods should provide a more powerful analysis of the processes described, and a

source for general theorems on syntactic structures.

8.2.2.5 Usefulness of SIGN

The schematic syntax notation offers only modest help with reasoning; it is hard to be aware of both the syntactic theory and its models at the same time. Elementary reasoning of the kind used in sketches is too difficult without computer aids, but the situation is better than other kinds of formal proof theory that have no diagrammatic help.

The current version of SIGN requires development into practical versions that limit the amount of reasoning expected. This task is of course exactly the kind that the AGENDA system is intended to help.

8.2.2.6 Feasibility of Notation Design

Would the ideas for notation design in Chapter 6 work? The research looks forward to notation design, but more detail is required before we will be able to judge whether the methods suggested would be workable. Full case studies of notations, especially in a software engineering context, are needed. It is not clear how practical it would be to design notation at this level of formality, and sophisticated support for reasoning would certainly be necessary. In any case, full notation design is expected to be a specialist task. There is no forseeable way to make this an easy task

At the outset of the research it was hoped that an approach to formal description of syntax would suffice to support notation design. It is evident that this cannot be accomplished without a universal theory of drawings, in the same way that linguistics is grounded in the study of phonetics and phonology. A theory of drawings would mainly concern the perception of space and recognition of shapes.

There is a long way to go before generic notation can be integrated with CASE methods. Integration would require standard data-formats for expressions, syntactic sketches and editor specifications. It would also require formal semantics for each notational role. Some important problems, such as translation, have been hardly touched upon.

8.2.2.7 Processing and Complexity

The question of how the syntactic specification gives rise to rules for interpreting drawings is an important one that is not fully addressed in the thesis. Part of the work of designing a notation is

to ensure that interpretation can be computed. We have seen in Chapter 4 that this is a matter of controlling the complexity of each layer in the tectonics (in terms of the size of sets constructed), but the details of how this might be done have not been given. The complexity of different kinds of sketch has not been related to the hierarchies that have been defined for grammars.

8.2.2.8 The Prototype

Is the AGENDA system feasible? The prototype development remains incomplete because certain theoretical problems (discussed in Chapter 6) were beyond the scope of the research effort. Exploring the prototype was effective in raising further areas for research, but the task of implementing a notation design tool was premature. Many questions are not addressed in the AGENDA system as described – the use of meta-schemas to define codices, for example, and making constructions on sketches to help with design of analogies.

How does the prototype reflect the research? The implementation follows the theoretical perspective as directly as possible. Where programming encountered difficulties, further theory was pursued to overcome them; often the solution was to express more of the structure as sketches.

What was the benefit of having the theoretical framework? Without a theory to work to, it would have been difficult to make any progress at all. The initial attempt to develop an editor for SIGN showed that an object-oriented programming system is suitable for this task, but requires a lot of work. As soon as the editor was rewritten to take account of the theory, the programming became more compact, and it was clear that the same techniques would suffice for any other notation – without claiming it to be the best approach in all cases.

8.2.3 Repeating the Attempt

How might the research be done on a second attempt? One surprise of the research was that the solutions found demanded such a deep appreciation of logic. It would be interesting to start from this direction, with a study of the development of practical systems of logic – especially intuitionist or constructivist logics and labelled deduction systems.

A better balance of theory and practice could be achieved by restricting the study to notations for a particular purpose, such as formal specification, or diagramming of functional programs. This would also provide more opportunity to investigate formal semantics.

Because of the importance of contributions from different disciplines, it would be best to pursue the research in a collaborative project involving cognitive psychologists, cognitive scientists, computational linguisticians, logicians, software developers and category theorists.

8.3 Further Work

The assessments in this chapter show that the thesis provides ample opportunities for further studies of notation and related areas. A list of possible future directions is commented upon below.

8.3.1 Practical Opportunities

Research consequent upon this thesis could attempt some of the following practical tasks :-

- R1 specifying a wide selection of notations by these methods in order to refine the theory
- R2 developing graphical theories suitable for particular classes of notations
- R3 re-design of the syntax formalism (SIGN) and variants in the light of practical experience
- R4 specifying standard notational mechanisms to build up a library resource
- R5 developing software for a notation design assistant
- R6 providing support for reasoning about syntax and calculating interpretations

A straightforward option [R1] would be a thesis devoted to specifying a variety of notations, while looking for cases that could cause difficulties, in order to refine the theory and method.

A thesis could investigate graphical theories for existing notations [R2] with a view to specifying the geometric elements that make up certain classes of diagrams.

Another option would be to explore whether the specification formalism could be improved, as Chapter 5 suggests, by creating variants of SIGN more suitable for practical use [R3] – or for different participants in software development. Applying the 'bootstrap' principle, variants could be designed by means of the techniques proposed in this thesis.

On a larger scale, a funded project could establish specifications and properties of familiar notational mechanisms [R4], which could become a resource for development of new practical notations and a medium for standards in graphical notation.

The larger project of building a full version of AGENDA is a natural successor to this research [R5]. This would need to follow up a number of strands made apparent by the endeavour of building

building the prototype in Chapter 7. For example, work is needed to investigate the relationship between geometry and syntax, in order to find suitable computational theories for drawings. Another concern is the question of how reasoning about syntax [R6] can be supported in practice. Other areas for practical research concern the problems of designing and supporting flexible protocols for editing, and the need for ways of offering easy routes for users to modify syntax.

8.3.2 Theoretical Opportunities

On the theoretical side, further research could address the following needs:-

R7	detailed study of how other approaches can be accommodated in this framework
R8	exploring fully the relationship between logical deduction and graph rewriting
R9	developing a universal pictorial theory that formalizes perception of drawings
R10	investigating the methodology of notation design
R11	analysing the mechanisms of metaphor in graphics
R12	extending the theory to cover ambiguity, approximation and vagueness in notations

We have noted a need for studying how the theoretical framework can accommodate the other approaches described in Chapter 3 [R7]. In this regard, the sketch formalism could be generalized to incorporate an ordering of syntactic sorts[1], in line with the techniques of computational linguistics and *object-oriented* notions of class-hierarchy and inheritance. Unifying all the approaches is a major task that would entail fundamental research on the relation between graph rewriting, unification, constraint logic, type theories and model theory [R8]. The aim would be to study how logics are implemented, in the sense of how reasoning is carried out within a calculus of limited resources. Such a project would necessarily rely on a collaboration of experts in the topics listed. The question of the hierarchy of complexity in logics and calculi could then be addressed, beyond the confines of grammar.

Theoretical methods are needed for implementing interpretive reasoning[2] by means of graph rewriting or other techniques.

> Interpretation is treated as incremental deduction of a semantic layer. Graph grammars are an attractive means of organizing this, because they implement a linear form of logic in which patterns of graphical items are premisses that are consumed when an inverse rule is applied, replacing the pattern

---

[1] This was also suggested in a communication from Prof. He Ji-Feng of Programming Research Group, Oxford University.

[2] – I.e. computational interpretation, which is different from *cognitive* interpretation in which there is always a synthesis of what is perceived and what is expected.

> by a higher syntactic item. This linearity is lost if we treat deduction as the inference of semantic properties from graphical ones, since each graphical property can be used many times in a proof. One of the features of linear deduction is the case of ambiguity between resources, which is not the same as the offering of alternatives or conjoint resources. Signification appears as a kind of linear implication.

The graphical theories of [R2] may help with a more difficult project, suggested in Chapter 6, of finding a formal theory of pictorial structure as perceived and understood by people [R9]. Qualitative topological theories are reported in (§3.2.2), but there is scope for many more notions. One mathematical topic that might find application is the theory of *matroids* (Welsh 1976, Oxley 1992). Matroids are 'pre-geometric' structures: abstract systems that have a concept of *dependency*, in the same sense as found in vector spaces.

> There are several reasons why this topic is attractive. Combinatoric structures such as graphs qualify as matroids, by defining an independent set of edges as one that contains no circuits (in a certain sense). The multidimensional variation found in formattable diagrams, with geometric constraints, can be described by matroids. One of the problems of diagrammatic design is in finding ways to express many dimensions of connectivity within a 2D graphical medium. In the semantics of expressions, notions of logical dependency occur; representing these in some kind of graphical dependency would appear to be an ideal way to assist reasoning in the subject domain.

The thesis invites a more general investigation [R10] of possible techniques and methods of notation design. This would include the quest for an understanding of the mechanisms of metaphor in the design of notations [R11], which is outlined in Appendix E. Research should study this approach in the light of the wide literature that exists on the subject of metaphor generally. Such a project would be more realistic if designed notations were tested with a suitable group of users.

Since this work has been arbitrarily restricted to notations whose coding is discrete, its application could be extended to include cases where continuous quantities are approximately represented [R12], such as geographical maps and architectural drawings. This could be approached within the same framework by using a medium such as topological spaces rather than sets. For the theory to treat ambiguity and vagueness in notations, it may be necessary to not only to consider other media, but to use a more general formulation of sketches. Although this is forseen in the sketch literature cited in Chapter 6, there would be less established support available. The work of (Goguen 1997) should also be taken into account.

8.4 Evaluation

In this last section we consider in what ways this research makes an original contribution to knowledge.

8.4.1 Originality and Usefulness

The originality of this work lies in newly applying mathematical concepts to address problems in a somewhat neglected area. This is the first study of notations to put forward clear mechanisms by which signification may take place. The study takes large steps towards formalizing semiotics and deriving a general theory of notation.

By providing a mathematical basis for notation structure, this work enables standard specifications of current software development notations to be constructed. The specifications do not rely on particular notions of grammar or spatial structure, as do other approaches. The formalism used is expressed in a new formal, graphical meta-notation based on the Theory of Sketches. As one of the few formalisms able to present specifications in diagrams, it is the first in which appearance of the specifying expressions is independent of that of the notation being defined. Sketches are shown to offer a hierarchy of logical strengths.

Although not a focus of the work, the theory makes possible a systematic analysis of analogy and metaphor in sign systems. These advances open up the possibility of giving full formal support to notation design, where support hardly exists at present. The work illustrates a novel approach to the development of generic editors.

8.4.1.1 Importance

How significant is this research? The thesis represents not only a step towards formalizing notations, but the beginning of a strategy for justifying notation design and finding new designs that will better serve the various purposes in software engineering – to help in thinking about system design problems and to communicate design decisions or proposals. For theorists, design of one's own notations is an important avenue for creative thought, especially in exploration of new topics.

The work draws attention to the nature of semiosis as a process that relies on varieties of tractable logic. This insight has repercussions throughout the subject of software development, since the

design of computer systems depends upon structures that can be understood and communicated by people, and is therefore a semiotic field. The ultimate benefit of formal approaches to notation will be seen as improved accuracy and effectiveness of software.

8.4.2 Conclusion

This thesis has demonstrated a coherent approach to formalizing graphical notations. It has shown that it is possible to assist the design of notations by means of diagrammatic expressions of syntax, with computational support. As a result, the work has uncovered a rich vein for future research, which has the potential to help in all activities of software engineering where notation plays a part.

Bibliography

Papers and Reports

Books and Proceedings

References: Papers and Reports

LNCS = Lecture Notes in Computer Science

Aït-Kaci, H. & Nasr, R. (1986) LOGIN: A Logic Programming Language with Built-in Inheritance. Journal of Logic Programming 3 p187-215. (Definite Clause)

Backlund, B.; Hagsand, O. & Pehrson, B. (1990) Generation of Visual Language-Oriented Design Environments. Journal of Visual Languages and Computing 1, 1990 p333-354.

Bagchi, Atish & Wells, Charles (1994) Varieties of Mathematical Prose. Case Western Reserve University, Cleveland Ohio.

Bagchi, A. & Wells, C. (1994) Graph-Based Logic and Sketches I: The General Framework. Case Western Reserve University.

Ballance, R.A.; Graham, S.L.; Van de Vanter, M.L. (1990) The Pan Language-Based Editor for Integrating Development Environments. ACM SIGSOFT Notes 15(6) Dec 1990 p77-93.

Balzer, R. & Goldman, N. (1985) Principles of Good Software Specification and their Implications for Specification Languages. In (Gehani & McGettrick 1985) p26-39

Banach, R. (1996) DPO Rewriting and Abstract Semantics via Opfibrations. SEGRAGRA'95 Workshop on Graph Rewriting and Computation, Italy Aug. 1995. Electronic Notes in Theoretical Computer Science 2.

Barr, Michael (1989) Models of Horn Theories. In (Gray & Scedrov 1989) p1-7.

Barr, M. & Wells, C. (1985) Toposes, Triples and Theories. Grundlehren der Mathematischen Wissenschaften 278. Springer.

Barwise, Jon (1993) Heterogeneous Reasoning. In: Mineau, G.; Moulin, B.; Sowa, J.F. (eds.) (1993) Conceptual Graphs and Knowledge Representation. Lecture Notes on Artificial Intelligence 754 Springer Verlag, p64-74.

Barwise, Jon & Etchemendy, John (1988) A Situation-Theoretic Account of Reasoning with Hyperproof. STASS meeting, Asilomar, March 1988.

Barwise, J. & Etchemendy, J. (1990a) Visual Information and Valid Reasoning. In: Zimmerman, W. (ed.) (1990) Visualization in Mathematics. Washington DC: Mathematical Association of America.

Barwise, J. & Etchemendy, J. (1990b) Information, Infons and Inference. In Cooper, Perry and Mukai (eds.) (1990) Situation Theory and its Applications. CSLI Lecture Notes Series 22. Stanford: CSLI publications.

Barwise, J. & Etchemendy, J. (1994) Hyperproof. Stanford: CSLI, and Cambridge University Press.

Barwise, J. & Etchemendy, J. (1995) Heterogeneous Logic. In: (Glasgow et al. 1995) p211-234. Also in (Allwein & Barwise 1996) p179-200.

Barwise, J. & Etchemendy, J. (1996) Computers, Visualization, and the Nature of Reasoning. To appear in Moor, James (ed.) On the Impact of Computers in Philosophy. Oxford: Blackwell.

Barwise, Jon & Hammer, Eric (1994) Diagrams and the Concept of Logical Systems. In Gabbay, D. (ed.) (1994) What is a logical system? Studies in Logic and Computation, Oxford University Press, p73-106. Also in (Allwein & Barwise 1996).

Barwise, J. & Perry, J. (1981) Situations and Attitudes. Journal of Philosophy 78(11) Oct.1981 p668-691.

Bastiani, A. & Ehresmann, C. (1973) Categories of Sketched Structures; Cahiers de Topologie et Geometrie Differentielle 13(73) p1-105.

Bauderon, M. (1996) Parallel Rewriting of Graphs through the Pullback Approach. SEGRAGRA'95 Workshop on Graph Rewriting and Computation, Italy Aug. 1995. Electronic Notes in Theoretical Computer Science 2, 1996.

Beer, S. & Welland, R. (1987) Software Design Automation in an IPSE. In (Nichols & Simpson 1987).

Black, W.J.; Sutcliffe, A.G.; Loucopoulos, P.; Layzell, P.J. (1987) Translation between Pragmatic Software Development Methods; in (Nichols & Simpson 1987) LNCS 289 p357-365. (AMADEUS unified conceptùal model; analyses JSD)

Bolognesi, T.; Hagsand, O. & Latella, D. (1991) The Definition of a Graphical G-LOTOS Editor using the Meta-Tool LOGGIE. Computer Networks and ISDN Systems 22(1) August 1991.

Brachman, R.J. & Schmolze, J.G. (1985) An Overview of the KL-ONE Knowledge Representation System. Cognitive Science Aug. 1985 p171-216.

Bricken, William & Gullischen, E. (1989) An Introduction to Boundary Logic with the Losp Deductive Engine. Future Computing Systems 2(4) 1989.

Broome, Paul & Lipton, James (1994) Combinatory Logic Programming: Computing in Relation Calculi. In: Bruynooghe, M. (ed.) Proc. International Logic Programming Symposium 1994, MIT Press. (Long version is available from the Logic and Computation Group, University of Pennsylvania.)

Burroni, A. (1991) Higher Dimensional Word Problem. In (Pitt et al. 1991) p94 - 105. (rewriting in n-categories)

Carpenter, R. (1995a) -- Interview published in: Ta! 3(1) Spring 1995. (Dutch student's magazine for computational linguistics)

Carpenter, R. (1991) Typed Feature Structures: A Generalization of First-Order Terms.

Carpenter, R. (1995b) The Turing-Completeness of Multimodal Categorial Grammars.

Cartmell, J.W. (1986) Formalising the Network and Hierarchical Data Models: An Application of Categorical Logic. In (Pitt et al. 1986) LNCS 240 p466-492.

Caswell, M.J.A. (1997) Equivalence of Formal Semantic Definition Methods. Formal Aspects of Computing 9 (1997) p68-77.

Chang, S.K. (1994) Ten Years of Visual Languages Research. IEEE Symposium on Visual Languages, 1994 p196-205.

Chok, S.S. & Marriott, K. (1995) Automatic Construction of User Interfaces from Constraint Multiset Grammars. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Citrin, W.; Doherty, M. & Zorn, B. (1994) Formal Semantics of Control in a Completely Visual Programming Language. IEEE Symposium on Visual Languages, St. Louis, Missouri, Oct.1994 p208-215.

Citrin, W.; Hall, R. & Zorn, B. (1995) Programming with Visual Expressions. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Clement, D.; Incerpi, J.; Kahn, G. (1990) CENTAUR: Towards a Software Toolbox for Programming Environments; in Software Engineering Environments 89; Fred Long ed; LNCS467 Springer 1990 p287-304.

Cohn, A.G. (1987) A More Expressive Formulation of Many-Sorted Logic. Journal of Automated Reasoning 3, 1987 p113-200.

Cohn, A.G. & Gooday, J.M. (1994) Defining the Syntax and Semantics of a Visual Programming Language in a Spatial Logic. AAAI'94 Spatial and Temporal Reasoning Workshop 1994.

Coleman, Edwin (1990) Paragraphy. Information Design Association Newsletter v6 n2 1990.

Cook, Steve & Masnavi, Siamak (1988) Visual Programming of User Interfaces. In: (Kilgour & Earnshaw 1988).

Coomber, C.J. & Childs, R.E. (1990) A Graphical Tool for the Prototyping of RTS; ACM SIGSOFT Notes 15(2) April 1990 p70-82.

Coppey, L. & Lair, C. (1984) Leçons de Théorie des Esquisses. Diagrammes 12, 1984.

Coppey, L. & Lair, C. (1988) Leçons de Théorie des Esquisses, Partie~II. Diagrammes 19, 1988.

Corradini, Andrea (1995) Concurrent Computing: from Petri Nets to Graph Grammars. SEGRAGRA'95 Workshop on Graph Rewriting and Computation, Italy Aug. 1995. Electronic Notes in Theoretical Computer Science 2.

Corradini, Andrea & Montanari, Ugo (1991) An Algebra of Graphs and Graph Rewriting; in (Pitt et al. 1991) LNCS 530 p236-260

Corradini, A.; Montanari, U.; Rossi, F.; Ehrig, H.; Löwe, M. (1991) Graph Grammars and Logic Programming. In: (Ehrig et al. 1991) LNCS 532 p221-237.

Courcelle, Bruno (1987a) A Representation of Graphs by Algebraic Expressions and its use for Graph Rewriting systems. In (Ehrig et al. 1987) LNCS 291 p112-132.

Courcelle, Bruno (1987b) On context-free Sets of Graphs and their Monadic Second-Order Theory. In (Ehrig et al. 1987) LNCS 291 p133-146.

Courcelle, Bruno (1990) Graph Rewriting: An Algebraic and Logic Approach; (van Leeuwen 1990) ch.5.

Courcelle, B. (1994) Monadic Second-Order Definable Graph Transductions, a Survey. Theoretical Computer Science 126 p53-75.

Courcelle, B. (1996) Basic Notions of Universal Algebra for Language Theory and Graph Grammars; To appear as a tutorial in TCS.

Davis, Alan M. (1988) A comparison of techniques for the specification of external system behaviour; Communications of ACM 31 (9) Sept 1988 p1098-1115.

de Rijke, Maarten (1992) The Modal Logic of Inequality. Notre Dame Journal of Formal Logic 57(2), 1992, p566-584.

Dörre, J. & Dorna, M. (1993) CUF: A Formalism for Linguistic Knowledge Representation. Deliverable R.1.2.A, DYANA 2, Aug. 1993.

Dörre, J.; König, E. & Gabbay, D.M. (1994) Fibred Semantics for Feature-Based Grammar Logic. Journal of Logic, Language and Information, Special Issue on Language and Proof Theory.

Egenhofer, M.J. (1991) Reasoning about Binary Topological Relations. In Günther, O. & Schek, H.-K. (eds) LNCS 525; Springer 1991. p143-160.

Ehrich, H.-D. & Lohberger, V.G. (1979) Constructing Specifications of Abstract Data Types by Replacements; in (Claus et al. 1979) p180-191

Ehrig, H.; Korff, M.; Lowe, M. (1991) Tutorial Introduction to the Algebraic Approach. In (Ehrig et al. 1991) p24-37.

Engels, G.; Lewerentz, C.; Schaeffer, W. (1987) Graph Grammar Engineering: A Software Specification Method; in (Ehrig et al. 1987) LNCS 291 p186-201.

Ferrucci, F.; Tortora, G.; Tucci, M. & Vitiello, G. (1994) A Predictive Parser for Visual Languages Specified by Relation Grammars. IEEE Symposium on Visual Languages, 1994 p245-252.

Ferrucci, F.; Tortora, G.; Tucci, M. & Vitiello, G. (1996) On the Generation and Recognition of Visual Languages: Relation Grammars and Related Approaches. In: TVL'96: International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Ferrucci, F.; Pacini, G.; Satta, G.; Sessa, M.; Tortora, G.; Tucci, M.; & Vitiello, G. (1996), Symbol-Relation Grammars: A Formalism for Graphical Languages. Submitted for publication.

Freyd, Peter (1972) Aspects of Topoi. Bulletin of the Australian Mathematical Society 7. p1-72.

Galton, A. (1988) Formal Semantics: Is it Relevant to Artificial Intelligence; AI Reviews 2(3) 1988

Gee, David M. (1995) Dept. of Computing, University of Nothumbria at Newcastle.

Gehani, Narain (1985) Specifications: Formal and Informal -- A Case Study; in (Gehani & McGettrick 1985) p173.

Girard, J.-Y. (1987) Linear Logic. Theoretical Computer Science 50, p1-102.

Godwin, W.H. (1991) Some Proposals Towards a Theory of Notation in Software Engineering. In De Neuman, Bernard et al. eds. (1991) Mathematical Structures for Software Engineering. IMA Conf. series 27, Oxford University Press, p53-82.

Goel, Vinod. (1992) "Ill-Structured Diagrams" for Ill-Structured Problems. In Proc. AAAI Symposium on Diagrammatic Reasoning, Stanford University, March 1992 p66-71.

Goguen, Joseph A. (1985) More Thoughts on Specification and Verification; in (Gehani & McGettrick 1985) p47.

Goguen, J.A. (1988) What is Unification?; Report SRI-CSL-88-2R2. Also in Nivat, M. & Aït-Kaci, H. (eds.) (1989) Resolution of Equations in Algebraic Structures, Vol. 1: Algebraic techniques. London: Academic Press, p217-261.

Goguen, J.A. (1997) Semiotic Morphisms. (Draft paper) Dept. of Computer Science and Engineering, University of California at San Diego.

Goguen, J.A & Burstall, R.M. (1984) Introducing Institutions. LNCS 164, Springer, Berlin.

Goguen, Joseph A. & Burstall, R.M. (1986) A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments. In (Pitt et al. 1986) LNCS 240 p313-333.

Goguen, J.A & Burstall, R.M. (1992) Institutions: Abstract Model Theory for specification and programming. Journal of ACM 39(1) p95-146.

Goguen, J.A. & Malcolm, G. (1997) A Hidden Agenda. Technical Report CS97-538, Dept. of Computer Science and Engineering, UCSD.

Goguen, J.A. & Meseguer, J. (1987) Models and Equality for Logic Programming. In Ehrig, H.; Kowalski, R. et al. eds (1987) Tapsoft'87 vol2, LNCS 250; Springer, p1-21.

Goguen, J. & Meseguer, J. (1989) Order-Sorted Algebra 1: Equational Deduction for Multiple Inheritance, Polymorphism, Overloading and Partial Operations. Tech. Report SRI-CSL-89-10, SRI International.

Goldblatt, R. (1986) Topoi: Categorical Analysis of Logic; Studies in Logic and Foundations of Mathematics 98, North Holland 1979 (2nd Edn. 1986).

Golin, E.J. (1991a) Parsing Visual Languages using Picture Layout Grammars. JVLC 2(4) Dec.1991 p371-393.

Golin, E.J. (1991b) A Method for the Specification and Parsing of Visual Languages. PhD Thesis, Brown University.

Golin, E.J.; Danz, S.; Larison, S.; Miller-Karlow, D. (1992) Pallette: An Extensible Visual Editor. In Proc. ACM / SIGAPP Symposium on Applied computing, March 1992 p1208-1216.

Gooday, J.M. & Cohn, A.G. (1996a) Visual Language Syntax and Semantics: A Spatial Logic Approach. International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Gooday, J.M. & Cohn, A.G. (1996b) Using Spatial Logic to Describe Visual Languages. Artificial Intelligence Review 1996 to appear.

Göttler, Herbert (1983) Attributed Graph Grammars for Graphics; in (Ehrig et al. 1983) LNCS 153 p130-142 (Nassi-Shneidermann example)

Göttler, H. (1987) Graph Grammars and Diagram Editing; in (Ehrig et al. 1987) LNCS 291 p216-231.

Göttler, H.; Günther, J. & Nieskens, G. (1990) Use Graph Grammars to Design CAD-Systems. In (Ehrig et al. 1991) LNCS 532 p396-410.

Gray, J.W. (1989a) The Integration of Logical and Algebraic Types. In Ehrig, H. et al. (eds) (1989) LNCS 393 Springer p16-35.

Gray, John W. (1989b) The Category of Sketches as a Model for Algebraic Semantics. In (Gray & Scedrov 1989) p109-136.

Green, T.R.G.; Petre, Marian; Bellamy, R.K.E. (1991) Comprehensibility of Visual and Textual Programs: The Test of Superlativism Against the "Match-Mismatch" Conjecture. In Empirical Studies of Programmers: Fourth Workshop, 1991 p121-146.

Gruzlewski, T. & Weiss, Z. (1991) Semantic Correctness of Structural Editing; ACM SIGPLAN 26(8) Aug1991 p111-120.

Gurr, C.A. (1996) On the Isomorphism (or Otherwise) of Representations. International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Guttag, John & Horning, J.J. (1985) Formal Specification as a Design Tool; in (Gehani & McGettrick 1985) p195

Haarslev, V. (1995) Formal Semantics of Visual Languages Using Spatial Reasoning. IEEE Symposium on Visual Languages, 1995 p156-163.

Haarslev, V. (1996a) A Fully Formalized Theory for Describing Visual Notations. International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Haarslev, V. (1996b) Formal Semantics of (Completely) Visual Languages.Tech. Rep. 1996 in preparation.

Haarslev, V. & Wessel, M. (1996) GenEd -- An Editor with Generic Semantics for Formal Reasoning about Visual Notations. Tech. Report in preparation, University of Hamburg Computer Science Dept.

Hammer E. (1993) Representing Relations Diagrammatically. In: (Allwein & Barwise 1993) p77-119.

Hammer E. (1993) Reasoning with Sentences and Diagrams. In: (Allwein & Barwise 1993) p120-143.

Hammer, Eric & Danner, N. (1996) Towards a Model Theory of Venn Diagrams. In (Allwein & Barwise 1996).

Hammer, Eric (1996) Peircean Graphs for Propositional Logic. In (Allwein & Barwise 1996).

Harel, David (1987) Statecharts: A Visual Formalism for Complex systems. Science of Computer Programming 8, 1987 p231-274.

Harel, David (1988) On Visual Formalisms; Communications ACM 31(5) May 1988 p514-531.

Harel, David; et al. (1990) STATEMATE: A Working Environment for the Development of Complex Reactive Systems; IEEE Trans.SE v16 n4 Apr 1990 p403-413.

Hekmatpour, S. (1990), Templa Graphica. New York, Prentice Hall.

Hekmatpour, S. & Woodman, M. (1987) Formal Specification of Graphical Notations and Graphical Software Tools. Open University CDFM Technical Report 87/7. In (Nichols & Simpson 1987) LNCS 289 p297-305.

Hekmatpour, S.; Preece, J.; Woodman, M.; Ince, D.C. (1988) Graphics Tools for Software Engineers. In (Kilgour & Earnshaw 1988)

Helm, R. & Marriott, K. (1991) A Declarative specification and semantics for visual languages. JVLC 2, 1991.+++

Hinton, G. (1979) Some Demonstrations of the Effects of Structural Descriptions in Mental Imagery. Cognitive Science 3 p231-250.

Hinton, G. (1980) Frames of Reference and Mental Imagery. In Long, J. & Baddeley, A. (Eds.) Attention and Performance. Hillsdale, New Jersey: Lawrence Erlbaum.

Hintikka, Jaakko (1979) Quantifiers in Natural Languages; in (Saarinen 1979) p81-117.

Hoare, C.A.R. (1986) The Mathematics of Programming; Clarendon.

Hull, M.E.C.; O'Donoghue, P.G.; Hagan, B.J. (1991) Development Methods for Real-Time Systems; Computer Journal v34 n2 April 1991; BCS CUP p164-172.

Ince, D.C. & Woodman, M. (1986) The Rapid Generation of a Class of Software Tools; Computer Journal 29(2) 1986, 151-160.

Indurkhya, Bipin (1992) Metaphor and Cognition. Studies in Cognitive Systems 13, Kluwer Academic.

Israel, D.J. & Brachman, R.J. (1984) Some Remarks on the Semantics of Representation Languages; in (Brodie et al. 1984) ch.5.

James, Jeffrey M. (1993) A Calculus of Number Based on Spatial Forms. MSc Thesis, University of Washington.

Jay, C.B. (1991) Tail Recursion from Universal Invariants. In (Pitt et al. 1991) LNCS 530 p151 - 163.

Jensen, Kurt (1991) Coloured Petri Nets: A High-Level Language for System Design and Analysis; in Rozenberg, Grz. ed. (1991) Advances in Petri Nets 1990; LNCS483 Springer, p342-416.

Jorge, J.A.P. & Glinert, E.P. (1995) Online Parsing of Visual Languages using Adjacency Grammars. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Kahn, K.M. & Saraswat V.A. (1990) Complete visualizations of Concurrent programs and their executions. IEEE Symposium on Visual Languages, Skokie, Illinois, Oct.1990 p7-14. [PJ]

Kamp, H. (1981) A Theory of Truth and Semantic Representation; in Groenendijk, J.; Janssen, T.; Stokhof, M.; eds (1981) Formal Methods in the Study of Language; Mathematical Centre Tracts 136, Amsterdam, p277-322.

Kauffman, Louis H. (1988) The Form of Arithmetic. In: Proceedings of the 18th International Symposium on Multiple-Valued Logic. IEEE Computer Society Press.

Kaul, M. (1982) Parsing of Graphs in Linear Time. In (Ehrig et al. 1983) LNCS 153 p206-218.

Kellman, P.J. & Shipley, T.F. (1991) A Theory of Visual Interpolation in Object perception. Cognitive Psychology 23, 1991 p141-221.

Kiesel, N.; Schürr, A. & Westfechtel, B. (1995) GRAS, A Graph-Oriented (Software) Engineering Database System. In information systems 20(1), Pergamon Press p21-52.

Kindfield, A.C.H. (1992) Expert Diagrammatic Reasoning in Biology. In Proc. AAAI Symposium on Diagrammatic Reasoning, Stanford University, March 1992 p41-46.

Klein, Ewan ed. (1987) Dialogues with Language, Graphics and Logic; ESPRIT 87, CEC-DGXIII North-Holland.

Kleyn, M.F. & Browne, J.C. (1993) A High Level Language for Specifying Graph Based Languages and their Programming Environments. In 15th International Conference on Software Engineering, May 1993 p324-335.

Knight, K. (1989) unification: A Multidisciplinary Survey. ACM Computing Surveys 21(1) p93-124.

Knuth, D.E. (1968) Semantics of Context-Free Languages; Mathematical Systems Theory 2(2) 1968 p127-145.

Koedinger K.R. (1992) Emergent properties and structural constraints: Advantages of diagrammatic representations for reasoning and learning. In: Narayanan, N. Hari (Ed.): AAAI Spring Symposium on Reasoning with Diagrammatic Representations. AAAI, Stanford, CA.

König, Esther. (1995) LexGram: A Practical Categorial Grammar Formalism. In: Proceedings of the Workshop on computational logic for natural language processing. Edinburgh, Scotland, April 1995. (cmp-lg/9504014). Also from Institute of Computational Linguistics, University of Stuttgart (1996 version).

Kuratowski, G. (1930) Sur le Probleme des courbes gauches en topologie. Fund. Math. 15-16, 1930, p271.

Kulpa, Z. (1994) Diagrammatic Representation and Reasoning. Machine Graphics and Vision 3(1/2) 1994 p77-103.

Lambek, J. (1958) The Mathematics of Sentence Structure. American Mathematical Monthly 65, 1958 p154-170. Reprinted in: Buszkowski, W.; Marciszewski, W. & van Benthem, J. (eds.) (1988) Categorial Grammar. John Benjamins, Amsterdam.

Lambek, J. (1961) On the Calculus of Syntactic Types. In: Jakobson, R. (ed.) (1961) Structure of Language and its Mathematical Aspects. Proc. Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island.

Larkin, J.H. & Simon, H.A. (1987) Why a Diagram is (Sometimes) Worth 10000 Words. Cognitive Science 11, p65-99.

Laurel, Brenda K. (1991a) Interface as Mimesis; in Laurel, B.K. ed. The Art of Human-Computer Interface Design, Addison Wesley. p67-85.

Lee, John; Oberlander, John; Stenning, Keith (1991) SIGNAL: Specificity of Information in Graphics and Natural Language. Report July 1991, Human Communication Research Centre, Edinburgh University.

Lemon, Oliver J. (1996) Semantical Foundations of Spatial Logics. Dept. of Computing Science, University of Manchester.

Lemon, Oliver J. (1996) Theories of Representation. In Logica'96 International Symposium, Prague. Filosophia Academic Publishing.

Levesque, H.J. (1988) Logic and the complexity of reasoning. Journal of Philosophical Logic 17 p355-389.

Loucopoulos, P. & Champion, R.E.M. (1990) Concept Acquisition and Analysis for Requirements Specification; SEJ v5 n2 Mar.1990 p116-124.

Lunney, T.F. & Perrott, R.H. (1988) Syntax-Directed Editing. Software Engineering Journal, March 1988 p37-46.

Makkai, Michael (1997) Sketches. Journal of Pure and Applied Algebra 115 (1, 2 & 3), 1997, (in three parts).

Makkai, M. & Reyes, G. (1977) First Order Categorical Logic. Lecture Notes in Mathematics 611, Springer.

Makkai, M. & Paré, R. (1990) Accessible Categories: the Foundations of Categorical Model Theory. Contemporary Mathematics 104; American Mathematical Society.

Marriott, K. (1994) Constraint Multiset Grammars. IEEE Symposium on Visual Languages, St. Louis, Missouri, Oct.1994 p118-125.

Marriott, K. & Meyer, B. (1996) Formal Classification of Visual Languages. International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

McWhirter, J.D. (1995) Characterization, specification and generation of visual language applications; PhD Thesis, Computer Science Dept., University of Colorado.

Menzies, T. (1995) Frameworks for Assessing Visual Languages; Tech. Report TR95-35, Dept. of Software Development, Monash University, Melbourne, Australia.

Meyer, B. (1992) Pictures Depicting Pictures – On the Specification of Visual Languages by Visual Grammars. Proc. IEEE Symposium on Visual Languages, Seattle, Sept. 1992 p41-48. (short version).

Minas, M. & Viehstaedt, G. (1995) DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Moher, T.G.; Mak, D.C.; Blumenthal, B.; Leventhal, L.M. (1993) Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In Empirical Studies of Programmers: Fifth Workshop, 1993, p137-161.

Montague, Richard (1974) The Proper Treatment of Quantification in Ordinary English; in (Thomason 1974): p247-270]

Moortgat, M. (1988) Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus. Gröningen-Amsterdam Studies in Semantics 9, Foris, Dordrecht.

Moortgat, M. (1994) Residuation in Mixed Lambek systems. Paper in the Deduction and Language Workshop, London. Manuscript: Institute for Speech and Language, OTS, Utrecht.

Morrill, Glyn (1993) -- Interview published in Dutch student's magazine for computational linguistics, Ta! 2(2) Summer 1993.

Morrill, G. (1994) Type Logical Grammar: Categorial Logic of Signs. Kluwer Academic, Dordrecht.

Müller, W. & Lehrenfeldt, G.(1994) Defining the Relational Grammar of Pictorial Janus; Cadlab Institute, Paderborn University, tech. rep. CR-07-94.

Myers, Brad A. (1988) The State of the Art in Visual Programming and Program Visualization; in (Kilgour & Earnshaw 1988)

Myers, B.A.; Guise, D.A. et al. (1990) Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. IEEE Computer 23(11) Nov. 1990 p71-85.

Mylopoulos, J. & Levesque, H.J. (1984) An Overview of Knowledge Representation; in (Brodie et al. 1984) p8.

Nagl, M.; Engels, G.; Gall, R.; Schaeffer, W. (1983) Software Specification by Graph Grammars; in (Ehrig et al. 1983) LNCS 153 p267-287.

Nagl, M. (1987) A Software Development Environment based on Graph Technology; in (Ehrig et al. 1987) LNCS 291 p458-478.

Nassi, I. & Shneiderman, B. (1973) Flowchart Techniques for Structured Programming; ACM SIGPLAN 8(8) Aug.1973 p12-26.

Netz, Reviel (1996) Greek Diagrams, their Use and their Meaning. Talk given at 3rd International Conf. for the History of Greek Mathematics. Delphi, August 1996.

Newberry, F.J. & Tichy, W.F. (1988) EDGE: An Extendible Graph Editor. Software -- Practice and Experience 20, June 1988 p63-88.

Nickerson, J. (1995) Visual Programming. PhD Thesis, UMI #9514409

Oberlander, J. & Stenning, K. (1990) Words, Picture and Calculi. Paper for 2nd Conference on Situation Theory and its Applications, Kinloch Rannoch, Scotland, Sept. 1990.

Palmer, Stephen (1992) Common Region: A New Principle of Perceptual Grouping. Cognitive Psychology 24, 1992 p436-447.

Pfeiffer, J.J. (1995) Ludwig2: Decoupling Program Representations from Processing Models. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Pineda, L.A. (1990) GRAFLOG: A Theory of Semantics for Graphics with Applications to Human-Computer Interaction and CAD Systems. PhD Thesis, Edinburgh University.

Pineda, L.A.; Klein, Ewan; Lee, John (1988) GRAFLOG: Understanding Drawings through Natural Language; Computer Graphics Forum 7 (1988) p97-103.

Pitts, Andrew M. (1989) Conceptual Completeness for First-Order Intuitionistic Logic: An Application of Categorical Logic. Annals of Pure and Applied Logic 41, 1989 p33-81.

Plotkin, G.D. (1981) A Structural Approach to Operational Semantics Report DAIMI FN-19, Computer Science Dept., Arhus University, Denmark.

Pollard, Carl J. & Sag, Ivan A. (1994) Head Driven Phrase Structure Grammar. University of Chicago Press.

Pong, M.C. (1991) I-Pigs: an Interactive Graphical Environment for Concurrent Programming; BCS Computer Journal 34(4) Aug 1991 p320-330.

Post, Julian (1986) Application of a Structured Methodology to a Real-time Industrial Software Development; SEJ Nov 1986 p222-247.

Pratt, T.W. (1971) Pair Grammars, Graph Languages and String-to-Graph Translation. Journal of Computer and system sciences 5, Academic Press p560-595.

Pratt, T.W. (1983) Formal Specification of Software using H-Graph Semantics; in (Ehrig et al. 1983) LNCS 153 p314-332.

Pratt, V.R. (1993) The Second Calculus of Binary Relations. In: Proceedings of MFCS'93, Gdansk, Poland 1993. LNCS 711, Springer Verlag. p142-155.

Pratt, V.R. (1995) Chu Spaces and their Interpretation as Concurrent Objects. In computer Science Today: Recent Trends and developments; LNCS 1000, Springer, Berlin, p392-405.

Pratt, V.R. (1988) Dynamic Algebras as a Well-Behaved Fragment of Relation Algebras. In Bergman, C.H.; Maddux, R.D.; Pigozzi, D.L. (eds.) (1990) Algebraic Logic and Universal Algebra in Computer Science. LNCS 425 Springer p77-110.

Randell, D.A. & Cohn, A.G. (1992) Exploiting Lattices in a Theory of Space and Time. In Lehmann, F. (ed.) (1992) Semantic Networks in Artificial Intelligence. Pergamon Press, Oxford. p459-476.

Reiss, S.P. (1987) Working in the Garden Environment for Conceptual Programming. IEEE Software Nov. 1987 p16-27.

Rekers, J. (1994) On the use of Graph Grammars for Defining the Syntax of Graphical Languages; Dept. of Computer Science, Leiden University Tech Report 94-11.

Rekers, J. & Schürr, A. (1995a) A Parsing Algorithm for Context-Sensitive Graph Grammars (long version). Technical Report 95-05, Leiden University.

Rekers, J. & Schürr, A. (1995b) A Graph Grammar approach to Graphical Parsing. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Roisin, J.-R. (1979) On Functorializing Usual First-Order Model Theory. In: Fourman, M.P.; Mulvey, C.J.; Scott, D.S. (Eds.) (1979) Applications of Sheaves. Proc. Symposium on Sheaf Theory, Durham 1977. Lecture Notes in Mathematics 753, Springer, p612-622.

Rounds, W.C. & Kasper, R.T. (1986) A Complete Logical Calculus for Record Structures Representing Linguistic Information. In: Proc. 15th Annual IEEE Symposium on Logic in Computer Science, Cambridge Massachusetts.

Rozenberg, G. & Welzl, E. (1986) Boundary NLC Graph Grammars -- Basic definitions, Normal Forms, and Complexity. information and Control 69, 1986 p136-167.

Sammet, Jean E. (1991) Some Approaches to, and Illustrations of, Programming Language History; Annals of the History of Computing v13 n1 1991 p33-50.

Schürr, A. (1990) PROGRES: A VHL-Language based on Graph Grammars. In (Ehrig et al. 1991) LNCS 532 Springer p641-659. Also Tech. report AIB 90-16, RWTH Aachen, Germany.

Schürr, A. (1994a) PROGRES: A Visual Language and Environment for Programming with Graph Rewriting; Tech. Report, Aachener Informatik-Berichte AIB 94-11.

Schürr, A. (1994b) Specification of Graph Translators with Triple Graph Grammars. Tech. Report, Aachener Informatik-Berichte AIB 94-12, RWTH Aachen. In Proc. WG'94 Int. Workshop on Graph Theoretic Concepts in Computer Science. LNCS 903, Springer-Verlag p151-163.

Schürr, A. (1994c) Logic Based Structure Rewriting Systems. In (Schneider & Ehrig 1994) LNCS 776 p341-357.

Schürr, A.; Winter, A. & Zündorf, A. (1995a) Visual Programming with Graph Rewriting Systems. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Schürr, A.; Winter, A. & Zündorf, A. (1995b) Graph Grammar Engineering with PROGRES. In Schäfer, A. & Botella, P. (eds.) (1995) Proc. 5th Eoropean Software Engineering Conf. ESEC'95, Barcelona, Sept. 1995, LNCS 989, Springer, Berlin, p219-234.

Serrano J.A. (1995) The Use of Semantic Constraints on Diagram Editors. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press. p211-216.

Serrano, J.A. & Welland, Ray (1997) VCT – A Formal Language for the Specification of Diagrammatic Modelling Techniques. Dept. of Computing Science, University of Glasgow, Scotland:

Shieber, Stuart M. (1986) An Introduction to Unification-Based Approaches to Grammar; CSLI Lecture Notes 4, University of Chicago Press.

Shin, Sun-Joo (1994) The Logical Status of Diagrams. Cambridge University Press.

Shin, Sun-Joo (1996) Situation-Theoretic Account of Valid Reasoning with Venn Diagrams. In (Allwein & Barwise 1996).

Shu, N.C. (1986) Visual Programming Languages: A Perspective and a Dimensional Analysis. In (Chang et al. 1986) p11-34.

Skousen, Royal (1975) Empirical Restrictions on the power of Transformational Grammars; in Storer, Th. & Winter, D. eds (1975) Formal Aspects of Cognitive processes; LNCS 22 Springer, p204-214.

Smolka, G. (1988a) A Feature Logic with Subsorts. LILOG-REPORT 33, IBM Stuttgart.

Smolka, G. (1988b) Logic Programming with Polymorphically Order-Sorted Types. LILOG-REPORT 55, IBM Stuttgart.

Sowa, J.F. (1979) Definitional Mechanism of Conceptual Graphs; in (Claus et al. 1979) p426-439

Stenning, Keith (1994) Logic as a Foundation for a Cognitive Theory of Modality Assignment [Allocation]. Report HCRC/RP-51, Human Communication Research Centre, Edinburgh University. Also in Masuch, M. (ed.) (1994) International Colloquium 'Logic at Work', Amsterdam University.

Stenning, K. & Oberlander, J. (1992) A Cognitive Theory of Graphical and Linguistic Reasoning: Logic and Implementation. HCRC / RP-20 Edinburgh University. Also in (1995) Cognitive Science 19, 1995 p97-140.

Stenning, K. & Tobin, R. (1994) Assigning Information to Modalities: Comparing Graphical Treatments of the Syllogism. HCRC RP-71, Human Communication Research Centre, Edinburgh University.

Sylva, L.; Freeman, E. et al. (1991) PRONET: The Application, and its use in Modelling the Uswest Telecommunications Network. Technical Report, USWest Advanced Technologies, Boulder, Colorado.

Tennant, N. (1986) The Withering Away of Formal Semantics. Mind and Language 4 p302-318.

Tse, T.H. & Pong, L. (1991) An Examination of Requirements Specification Languages; Computer Journal v34 n2 April 1991; BCS CUP p143-152.

Tse, T.H. & Pong, L. (1989) Towards a Formal Foundation for De Marco Data Flow Diagrams; Computer Journal 32(1) 1989 p1-12.

Üsküdarli, S.M. (1994) Generating Visual Editors for Formally Specified Languages. Proc. 1994 IEEE Symposium on Visual Languages, p278-285.

Üsküdarli, S.M. (1995) Specifying Visual Syntax. In ASD+SDF'95: A Workshop on Generating Tools from Algebraic Specifications, May 1995.

Üsküdarli, S.M. & Dinesh, T.B. (1995a) Towards a VP Environment Generator for Algebraic Specifications. IEEE Symposium on Visual Languages, Darmstadt, Sept.1995; IEEE Computer Society Press.

Üsküdarli, S.M. & Dinesh, T.B. (1995b) VODL: A picture description language. Tech. report, University of Amsterdam. To Appear.

Varadharajan, V. & Baker, K.D. (1987) Directed Graph Based Representation for Software System Design; Software Engineering Journal Jan. 1987.

Varela, F.J. (1975) A Calculus for Self-Reference. International Journal of General Systems 2, 1975 p5-24.

Varela, F.J. (1979) Principles of Biological Autonomy. New York: North Holland.

Viehstaedt, G. (1995) A generator for diagram editors. PhD Thesis, University of Erlangen.

Viehstaedt, G.; Minas, M. (1995) Graphical Representation and Manipulation of Complex Structures Based on a Formal Model. In Iivari, J.; Lyytinen, K.; Rossi, M. (eds.), Proc. Advanced Information Systems Engineering, 7th International Conference, CAiSE'95, Jyväskylä, Finland, June 1995. Lecture Notes in Computer Science 932, Springer Verlag, Heidelberg, p243-254.

Waite, Kevin W. (1988) Visualising Abstract Data Types; in (Kilgour & Earnshaw 1988)

Wang, D. (1995) Studies on the Formal Semantics of Pictures. PhD Thesis, University of Amsterdam.

Wang, D.; Lee, J.R. & Zeevat, H. (1995) Reasoning with Diagrammatic Representations. In (Glasgow et al. 1995) p339-393.

Wang, D. & Zeevat, H. (1996) A Syntax Directed Approach to Picture Semantics. International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Ward, P. (1986) The Transformation Schema: an Extension of the Data Flow Diagram to Represent Control and Timing Information; IEEE S-E 12(2) Feb1986 p198-210.

Weitzmann, L. & Wittenburg, K. (1994) Automatic Presentation of Multimedia Documents using Relational Grammars. Proc. 2nd ACM Multimedia Conference, San Francisco Oct.1994 p443-451.

Welland, Ray; Beer, Stephen & Sommerville, Ian (1990) Method Rule Checking in a Generic Design Editing System. Software Engineering Journal 5(2) March 1990, 105-115.

Wells, C. (1994) Communicating Mathematics: Useful Ideas from Computer Science. Case Western Reserve University, Cleveland Ohio.

Wells, C. (1990) A Generalization of the Concept of Sketch; TCS 70, 1990; p159-178.

Wells, C. (1994) Sketches: Outline with References. Report, Case Western Reserve University, Cleveland Ohio.

Wells, C. & Barr, M. (1987/8) The Formal Description of Data Types using Sketches. In Main, M. et al. (ed.) (1988) Mathematical Foundations of Programming Language Semantics. LNCS 298, Springer.

Winskel, G. (1988) An Introduction to Event Structures. In de Bakker, de Roever, Rozenberg (eds.) Linear Time, Branching Time and Partial Order Logics and Models of Concurrency. LNCS 354 Springer Verlag p364-397.

Wittenburg, K. (1993) Adventures in Multi-dimensional Parsing: Cycles and Disorders. Int. Workshop on Parsing Technologies, Netherlands & Belgium Aug. 1993.

Wittenburg, K. & Weitzmann, L. (1996) Relational Grammars: Theory and Practice in a Visual Language Interface for Process Modelling. In: Proceedings AVI/TVL'96: International Workshop on the Theory of Visual Languages, Gubbio, Italy, May 1996.

Wittenburg, K.; Weitzmann, L.; Talley, J. (1991) Unification-based Grammar and Tabular Parsing for Graphical Languages. JVLC 2(4) Dec.1991 p347-370.

Woodman, M.; Ince, D.C.; Preece, J.; Davies, G. (1986) A Grammar Formalism as a basis for the Syntax-directed Editing of Graphical Notations; Open University CDFM Technical Report 86/19.

Zave, Pamela (1985) An Operational Approach to Requirements Specification for Embedded Systems. In (Gehani & McGettrick 1985) p148.

Zave, Pamela & Yeh, R.T. (1985) Executable Requirements for Embedded Systems. In (Gehani & McGettrick 1985) p341,2.

Books and Proceedings

Allwein, G. & Barwise, J. (Eds.) (1993) Working Papers on Diagrams and Logic. Preprint IULG-93-24, Logic Group, Indiana University, Bloomington, IN.

Allwein, Gerald & Barwise, John (1996) Logical Reasoning with Diagrams; Studies in Logic and Computation 6; Oxford University Press USA

Barthes, Roland (1967) Elements of Semiology; Cape.

Berge, Claude (1970) Graphs and Hypergraphs; North Holland.

Bergstra, J.; Heering, J. & Klint, P. (eds.) (1989) Algebraic Specification. ACM Press, Frontier series, with Addison Wesley. (ch1, 6).

Bertin, Jacques (1982) Graphics and Graphic Information Processing; de Gruyter.

Bertin, J. (1983) Semiology of Graphics: Diagrams, Networks, Maps. University of Wisconsin Press.

Bird, Richard & Wadler, Philip (1988) Introduction to Functional Programming. Prentice Hall.

Brodie, M.L.; Mylopoulos, J.; Schmidt, J.W. (1984) On Conceptual Modelling; Springer.

Cajori, Florian (1929) A History of Mathematical Notations, Vol II ; Open Court, Chicago. (also published 1974).

Carpenter, R. (1996) Type-Logical Semantics. MIT Press.

Chang S.-K., Ichikawa T., Ligomenides P.A. (Eds.) (1986) Visual Languages. Plenum Press, New York.

Checkland, Peter (1981) Systems Thinking, Systems Practice. New York: John Wiley & Sons.

Chen, Peter P. (1978) The Entity-Relationship Approach to Logical Database Design. QED Monograph Series 6. Massachusetts: Information Services Inc.

Chomsky, Noam (1965) Aspects of the Theory of Syntax; MIT Press.

Claus, V.; Ehrig, H.; Rozenberg, G. eds. (1979) Graph Grammars and their Application to Computer Science & Biology: (Proc. 1st Int. Workshop). Lecture Notes on Computer Science 73. Heidelberg: Springer.

Clocksin, W.F. & Mellish, C.S. (1987) Programming in Prolog. New York: Springer.

Cohen, B.; Harwood , W.T.; Jackson, M.I. (1989) The Specification of Complex Systems. Addison-Wesley. (p94-100).

Dalrymple, M.; Kaplan, R.M.; Maxwell, J.T.; Zaenen, A. (eds.) (1996) Formal Issues in Lexical-Functional Grammar. CSLI Lecture Notes 47. Cambridge University Press.

De Marco, T. (1978) Structured Analysis and System Specification; Yourdon NY.

Deransart, Pierre; Jourdan, M.; Lorho, B. (1988) Attribute Grammars. LNCS 323, Springer.

Dershowitz, N. (1989) Rewriting Techniques and Applications VII. LNCS355, Springer.

Devlin, Keith J. (1991) Logic and Information. Cambridge University Press.

Dijkstra, Edsger W. (ed.) (1990) Formal Development of Programs and Proofs. Addison Wesley

Eco, Umberto (1976) A Theory of Semiotics; Indiana University Press.

Eco, Umberto (1984) Semiotics and the Philosophy of Language. London: Macmillan.

Ehrig, H.; Nagl, M.; Rozenberg, G. eds (1983) Graph Grammars and their Application to Computer Science: (Proc. 2nd Int. Workshop). LNCS153; Springer.

Ehrig, H.; Nagl, M.; Rozenberg, G.; Rosenfield, A. (eds.) (1987) Graph Grammars and their application to Computer Science (3rd). LNCS 291, Springer.

Ehrig, H.; Kreowski, H.-J.; Rozenberg, G. eds (1991) Graph Grammars and their application to Computer Science (4th). LNCS 532, Springer.

Ehrig, H.; Engels, G.; Rozenberg, G. eds. (1995) Graph Grammars and their Application to Computer Science: (Proc. 5th Int. Workshop). LNCS 1073; Springer.

Ellis, G. & Levinson, R. (Eds.) (1992) Proceedings of the 1st International Workshop on Peirce: A Conceptual Graphs Workbench", Las Cruces, New Mexico, July 1992.

Ellis, Gerard & Levinson, Robert (Eds.) (1994) Proceedings of the 3rd International Workshop on Peirce: A Conceptual Graphs Workbench, University of Maryland, August 1994.

Ellis, Gerard; Levinson, R.; Rich, W.; Sowa, John F. (Eds.) (1995) Conceptual Structures: Applications, Implementation and Theory, Proceedings of Third International Conference on Conceptual Structures, ICCS'95, Santa Cruz, CA, USA, August 1995. Lecture Notes in AI 954, Springer-Verlag.

Findler, N.V. (1979) Associative Networks: Representation and Use of Knowledge by Computer; Academic.

Frege, Gottlob (1972) Conceptual Notation and Related Articles. Oxford: Clarendon Press.

Freyd, Peter & Scedrov, A. (1990) Categories and Allegories. North-Holland.

Gane, C. & Sarson, T. (1977) Structured Systems Analysis: Tools and Techniques. IST, NY.

Gehani, N. & McGettrick, A.D. eds. (1985) Software Specification Techniques. Addison Wesley.

Girard, J-Y.; Lafont, Yves; Taylor, Paul (1989) Proofs and Types. Cambridge University Press.

Glasgow, J.I. & Narayanan, N.H. & Chandrasekaran, B. (eds.) (1995) Diagrammatic Reasoning: Cognitive and Computational Perspectives. Menlo Park: AAAI Press / Cambridge, MA: MIT Press.

Goodman, Nelson (1968) Languages of Art: An approach to a Theory of Symbols. Indianapolis: Bobbs-Merrill.

Gordon, M.J.C. (1979) The Denotational Description of Programming Languages: An Introduction. Springer-Verlag.

Gorny, P. & Tauber, M.J. (1987) Visualization in Programming. LNCS 282, Springer.

Gray, John W. & Scedrov, Andre (eds.) (1989) Categories in Computer Science and Logic: (proc. conf. June 87) Contemporary Mathematics 92; American Mathematical Soc.

Hammer, Eric (1996) Logic and Visual Information. Studies in Logic, Language and Information 3. CSLI, Cambridge University Press.

Hardwick, C.S. (ed.) (1977) Semiotics and Significs: The Peirce/ Welby Correspondence; University Press. (pictorial logic)

Harland, David (1984) Polymorphic Programming Languages. Ellis Horwood, Chichester UK.

Harland, David (1986) Concurrency and Programming Languages. Ellis Horwood, Chichester UK.

Hartshorne, C. & Weiss, P. eds. (1933) The Collected Papers of Charles Sanders Peirce. Harvard University Press, vols 3, 4. (v3 Exact Logic; v4 The Simplest Mathematics)

Hartshorne, C. & Weiss, P. (eds.) (1933) The Collected Papers of Charles Sanders Peirce; Harvard University Press.

Hintikka, J. (1969) Models for Modalities; D. Reidel, Dordrecht.

Houser, N.; Roberts, D.D. & van Evra, J. (eds.) (1997) Studies in the Logic of C. S. Peirce. Bloomington: Indiana University Press.

Jackson, M.A. (1983) System Development. Prentice-Hall.

Johnston, David E. & Postal, Paul M. (1980) Arc Pair Grammar. Princeton University Press.

Jones, Cliff B. (1990) Systematic Software Development using VDM. Prentice-Hall (2nd edn.)

Jouannaud, J.P. ed. (1985) Rewriting Techniques and Applications V, LNCS202, Springer.

Kilgour, A.C. & Earnshaw, R.A. eds (1988) Graphics Tools for Software Engineering: Visual Programming & Program Visualisation (Proc BCS Symposium London March 1988). British Computer Society, London.

Kilgour, A.C. & Earnshaw, R.A. eds (1988) Graphics Tools for Software Engineering: Visual Programming & Program Visualisation (Proc BCS Symposium London March 1988). British Computer Society, London.

Lambek, J. & Scott, P.J. (1986) Introduction to Higher Order Categorical Logic. Cambridge University Press.

Laurel, B.K. (1991b) Computers as Theatre. Addison Wesley

Lescanne, P. ed. (1987) Rewriting Techniques and Applications VI. LNCS 256, Springer.

Levinson, R. & Ellis, G. (Eds.) (1993) Proceedings of the 2nd International Workshop on Peirce: A Conceptual Graphs Workbench. Quebec City, Canada, August 1993.

MacLane, Saunders (1971) Categories for the Working Mathematician. Springer.

Manzano, Maria (1996) Extensions of First-Order Logic. Cambridge Tracts in Theoretical Computer Science 19, Cambridge University Press.

Martin, J. & McClure, C. (1985) Structured Techniques for Computing. Prentice-Hall.

Martin, R.M. (1978) Semiotics and Linguistic Structure – A Primer in Philosophic Logic. State University of NY, Albany. (Formal logical analysis of the use of formal language)

MASCOT (1987) The Official Handbook of MASCOT; Crown, HMSO, UK.

Milner, R. & Strachey, C. (1976) A Theory of Programming Language Semantics. Chapman and Hall.

Nichols, H.K. & Simpson, D. eds (1987) Proc. ESEC87: First European Software Engineering Conf. LNCS 289, Springer.

Oxley, J.G. (1992) Matroid Theory. Oxford University Press.

Peirce, C.S. (1984) Writings of C.S. Peirce. Indiana University Press

Peterson, J.L. (1981) Petri Net Theory and the Modelling of Systems. Prentice-Hall.

Peyton Jones, Simon (1987) Implementation of Functional Languages. Prentice-Hall. (G-machine)

Pitt, D.H. et al. (eds.) (1986) Category Theory and Computer Science. LNCS 240 Springer.

Pitt, D.H. et al. (eds.) (1991) Category Theory and Computer Science. LNCS 530 Springer.

Potter, B.; Sinclair, J.; Till, D. (1991) An Introduction to Formal Specification and Z. Prentice-Hall.

Roberts, Don D. (1973) The Existential Graphs of Charles S. Peirce. The Hague: Mouton.

Rydeheard, D.E. & Burstall, R.M. (1988) Computational Category Theory. Prentice-Hall.

Saarinen, Esa (ed.) (1979) Game-Theoretical Semantics; D. Reidel, Dordrecht.

Saussure (1916) Cours de Linguistique General; engl. tr. Baskin,W. (1950). Peter Owen.

Shannon, Claude & Weaver, Warren (1964) Mathematical Theory of Communication. Illinois.

Seligman, Jerry & Westerstahl, Dag (eds.) (1996) Logic, Language and Computation vol.1. CSLI Lecture Notes, Cambridge University Press.

Sowa, John F. (1984) Conceptual Structures: Information Processing in Mind and Machine; Addison Wesley.

Sowa, J.F. (ed.) (1991) Principles of Semantic Networks: Explorations in the Representation of Knowledge. San Mateo, California: Morgan Kaufmann.

Spencer-Brown, George (1969) Laws of Form. New York: Bantam. (Also published by New York: Julian Press 1972).

Spivey, J.M. (1988) Understanding Z. Cambridge University Press.

Tepfenhart, W.M.; Dick, J.P.; Sowa J.F. (Eds.) (1994) Conceptual Structures: Current Practices. Proceedings of Second International Conference on Conceptual Structures: ICCS '94, College Park, Maryland, August 1994. Lecture Notes in AI 835, Springer-Verlag.

Thomason, Richard (ed.) (1974) Formal Philosophy: Selected Papers of Richard Montague; Yale University Press.

Tufte, E.R. (1990) Envisioning Information. Graphics Press, Cheshire, CT.

van Leeuwen, Jan; ed. (1990) Formal Models and Semantics (Handbook of Theoretical Computer Science vol B); Elsevier Science.

Vickers, Stephen (1989) Topology via Logic. Cambridge University Press

Watt, D.A. (1991) Programming Language Syntax and Semantics. Prentice Hall.

Welsh, D.J.A. (1976) Matroid Theory. London: Academic Press.

Whitehead, Alfred North (1948) An Introduction to Mathematics. New York: Oxford University Press.

Yourdon, E. & Constantine, L. (1978) Structured Design. Yourdon, NY.

Zhao, R. (1993) Handsketch-based Diagram Editing. Teubner, Stuttgart.

Appendices

A. A Note on Syntactic Symmetry

B. A Sketch for a BNF Grammar

C. A Proof Concerning a Simple Rewrite Rule

D. A Short Glossary

E A Note on Sketching Metaphors

F. Smalltalk *Classes* of the Prototype Implementation

Appendix A.

A Note on Syntactic Symmetry

This short note discusses and illustrates the notion of symmetry in forms and sketches, which relates to the concept of geometric shape either in a graphical or more abstract sense.

A.1 Syntactic Symmetry and Abstract Geometry

Abstract ideas of shape can be described by algebraic operations in geometry. We examine the case where the items or expressions of a notation are symmetric in some way.

In the language of algebra, a morphism from an entity to itself is called an *endomorphism*; if it is an isomorphism, it is called an *automorphism*. The associative composition of morphisms provides a total binary operation on the generated set of endomorphisms on an entity, comprising the identity morphism and all composites, which means that the set carries an algebraic structure known as a *monoid*. The subset of automorphisms is closed under composition, and has an inverse for each member, which makes it a *group*.

Forms that are invariant under a group of operations are said to possess the global property of *symmetry*. Symmetries of this kind may be evident at any level of syntax, but are especially noticeable in graphical properties. Automorphisms within a sketch show where symmetries can be drawn into the graphics.

A Symmetric Relation

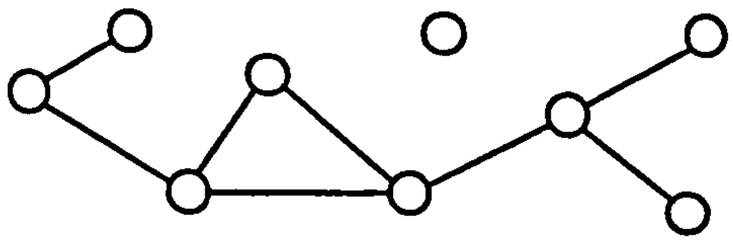


Figure A.1 A web

For example, a non-directed graph (or *web*) [fig A.1] may be used to denote a symmetric binary relation. We consider the sketch for a 'nearness' relation, which describes a subset of ordered pairs:-

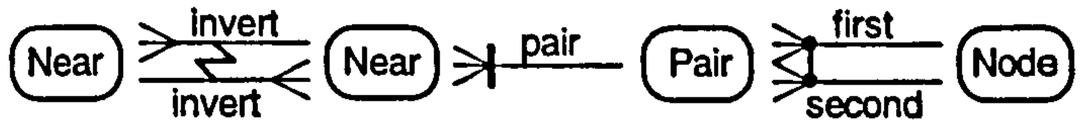


Figure A.2 A sketch for a relation on nodes

In this sketch, it is intended that a web is ambiguous as to which end of any arc is first and which is second. The map *invert* associates each *near*-arc of a graph with an arc in the opposite direction, by virtue of the equality:-

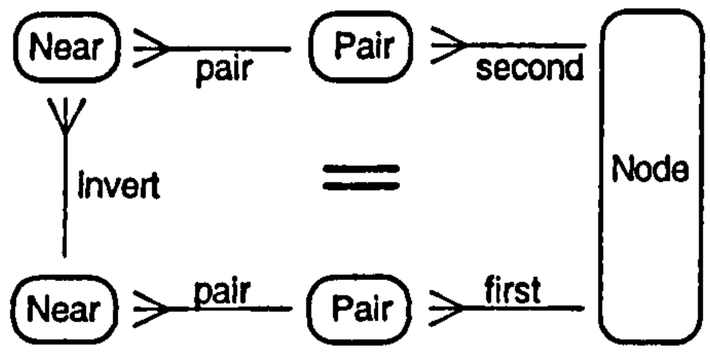


Figure A.3 An equality of arc-inversion

From the sketch,

```
invert: Near → Near
invert;invert = Near
invert;pair;first = pair;second
```

Hence

```
invert;invert;pair;first = invert;pair;second
pair;first = invert;pair;second
```

This abstract symmetry can then be realized through geometric symmetry. The pictorial link-items that express nearness of two nodes may be symmetric, with *invert* corresponding to rotation through 180°.

A.2 Symmetric Sketches

Symmetries may also exist within sketches (e.g. chains in [fig 5.26] of §5.3.2.2). We may define:-

An *endo-codex* is a codex from a sketch to itself; it is an *auto-codex* if it is an isomorphism.

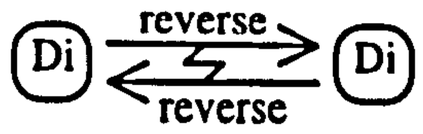
The set of endo-codices generated by composition of endo-codices on a sketch has the structure of a *monoid*. The subset of autocodices is a *group*.

Since a codex re-interprets one sketch in another, we can follow a geometric metaphor and regard the group of auto-codices as giving all viewpoints on any expression. If all expressions are symmetric under a group of auto-codices, then viewpoints in the group have no effect on meaning.

Let *Di* be the sketch for directed graphs:

.

There is an auto-codex on this sketch which swaps the roles of its two maps, and thus has the effect of reversing the arcs of a graph. This codex (*reverse*) is self-inverse, as depicted in the meta-schema:



If *g* is a formal digraph (a model of *Di*), then `reverse;g` is the 'same' graph but with arcs reversed.
A graph *g* is symmetric under reversal iff `reverse;g ≅ g` .

As with the digraph, the nearness sketch above also exhibits symmetry between the maps *first* and *second*. The autocodex that swaps these maps represents the semantic operation of inverting the relation. In the case of such a self-dual graph, this global reversal cannot affect meaning.

Appendix B.

An Example Sketch for a BNF Grammar

This example explores how a conventional BNF syntax can be specified in SIGN. This may be compared with the general case of sketches for context-free grammars given in (Wells & Barr 1988), in which the initial algebra for a suitable sketch is the set of derivation trees for a context-free language.

B.1 Sketches for a Syntax of Boolean Expressions

The syntactic fragment covers expressions such as:

```
¬ (T ∨ (¬ (F ∧ T)))
```

in which there is one unary and two binary operators, and two constants.

The 'tagmatic' syntax may be defined by the BNF rules:

Bool ::= Sent | Value

Char ::= Value | Op | Not | Brac | Ket

Sent ::= Dyad | Neg

Dyad ::= Part Op Part

Neg ::= Not Part

Part ::= Value | Sub

Value ::= True | False

Sub ::= Brac Sent Ket

Op ::= And | Or

– where Bool is the type of a complete formula and Char is the type of a character in the formula. Note that the characters themselves, which are terminal symbols, are here represented by the names:

Not And Or Brac Ket True False.

This avoids extending the BNF formalism with extraneous literal characters, and emphasizes the fact that character-shapes are not formally specified.

The schemas below are intended to define the lexical and tagmatic layers of syntax – neither stating how the characters are drawn, nor interpreting the expression into a normal form or evaluation. The schemas may define the abstract shape of expressions as strings of characters, which is taken for granted in BNF. Strings are defined as certain connected directed graphs

whose arcs are characters, each linking a node on the left to one on the right, as described in Chapter 5.

If we depict the BNF rules directly, we get schemas such as the following:-

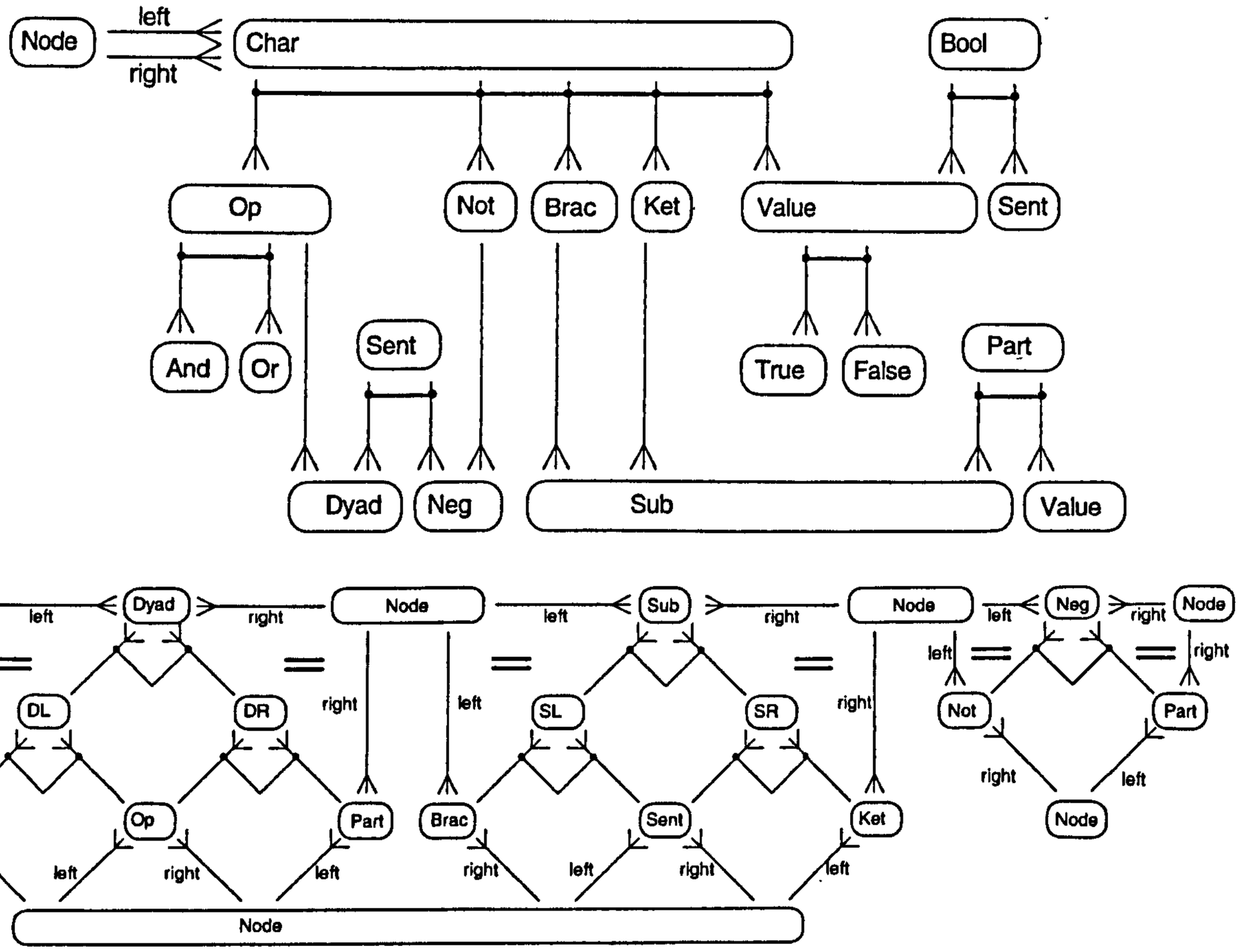


Figure B.1 Schemas for generative Boolean syntax

Apart from *Node*, every entity is a type of substring between a left node and a right node. The substrings are defined as limits – for example *Neg* is a pullback: the apex of a cone on base:

```
(Not → Node ← Part)
```

that denotes a *Not*-character preceding a *Part*-phrase.

In the case of *Dyad*, three pullbacks are used to construct a cone on base:

```
(Part → Node ← Op → Node ← Part).
```

B.1.1 A Notation for Map-Labelling

In this example the maps are given structured names rather than the unique labels of SIGN in Chapter 5. Map labels in the schemas here need not be unique, but maps with the same label and same domain entity must be equal. Where no name is shown, the map takes the default name of its target-entity, but uncapitalized. Thus the map from *Value* to *Char* is called "*char*".

To avoid ambiguity, paths are given unique names, built as follows:

<name of start entity> followed by a list of <map-label>, separated by stops.

– if two paths have the same name, they are equal.

E.g. the left-node for a value is identified by the pathname `Value.char.left`,
the right-node for a left-bracket (*Brac*) is identified by the pathname `Brac.char.right`, and
the left-node for a right-bracket (*Ket*) is identified by the pathname `Ket.char.left`.

A map can then be uniquely named by its singleton path, e.g. `Value.char`, and an identity map on an entity is named (as before) by its entity name, which also denotes an empty path.

The equalities shown in [fig B.1] can then be written:-

```
Dyad.left = Dyad.dl.part.left
Dyad.right = Dyad.dl.part.right
```

E.g. the first of these equalities has the effect of constructing *Dyad.left* as the composite:

```
Dyad.dl;DL.part;Part.left
```

of functions denoted by the maps.

By exploiting the algebraic definition of theory-categories, derived maps can generally be named by textual expressions. For example, we can calculate the bounding nodes of a formula in Bool as follows:

```
Bool.left = Value.char.left ∨ Sent.left
```

– which constructs the unique map from Bool, which is the apex of cocone on base (*Value Sent*). The symbol /∨/ signifies choice between disjoint cases. (To avoid ambiguity it may sometimes be necessary to label the cones and co-cones of the sketch.)

Using this notation, the rest of the calculation for bounding nodes of a formula in Bool is as follows:

```
Sent.left = Dyad.left ∨ Neg.left
Neg.left = Neg.not.char.left
Dyad.left = Dyad.dl.part.left
Part.left = Value.char.left ∨ Sub.sl.brac.char.left
Op.left = And.char.left ∨ Or.char.left
```

Similarly,

```
Bool.right = Value.char.right ∨ Sent.right
Sent.right = Dyad.right ∨ Neg.right
Neg.right = Neg.part.right
Dyad.right = Dyad.dr.part.right
Part.right = Value.char.right ∨ Sub.sr.ket.char.right
Op.right = And.char.right ∨ Or.char.right
```

[Some of these equalities are omitted from the above schema, for brevity.]

The full sketch describes the steps in parsing any string of characters in search of substrings of type Bool. It specifies constraints on the search, but does not lead to a direct construction of the maps `Bool.left` and `Bool.right` – owing to cyclic dependencies in the constraints.

B.1.2 When is a string well-formed?

If we wish to specify that an expression must consist solely of a set of boolean formulas, it is necessary to add several more constraints.

It is clear that the following isomorphisms hold:

```
Dyad ≅ DL ≅ DR ≅ Op
Sub ≅ SL ≅ SR ≅ Brac ≅ Ket
Neg ≅ Not
```

We need to constrain every string of characters to be an item with property *Bool*, though we note that substrings may also be of type *Bool*. This makes it possible to simplify the sketch, along the lines of [fig B.2], with further constraints to ensure every string defines a formula of Bool.

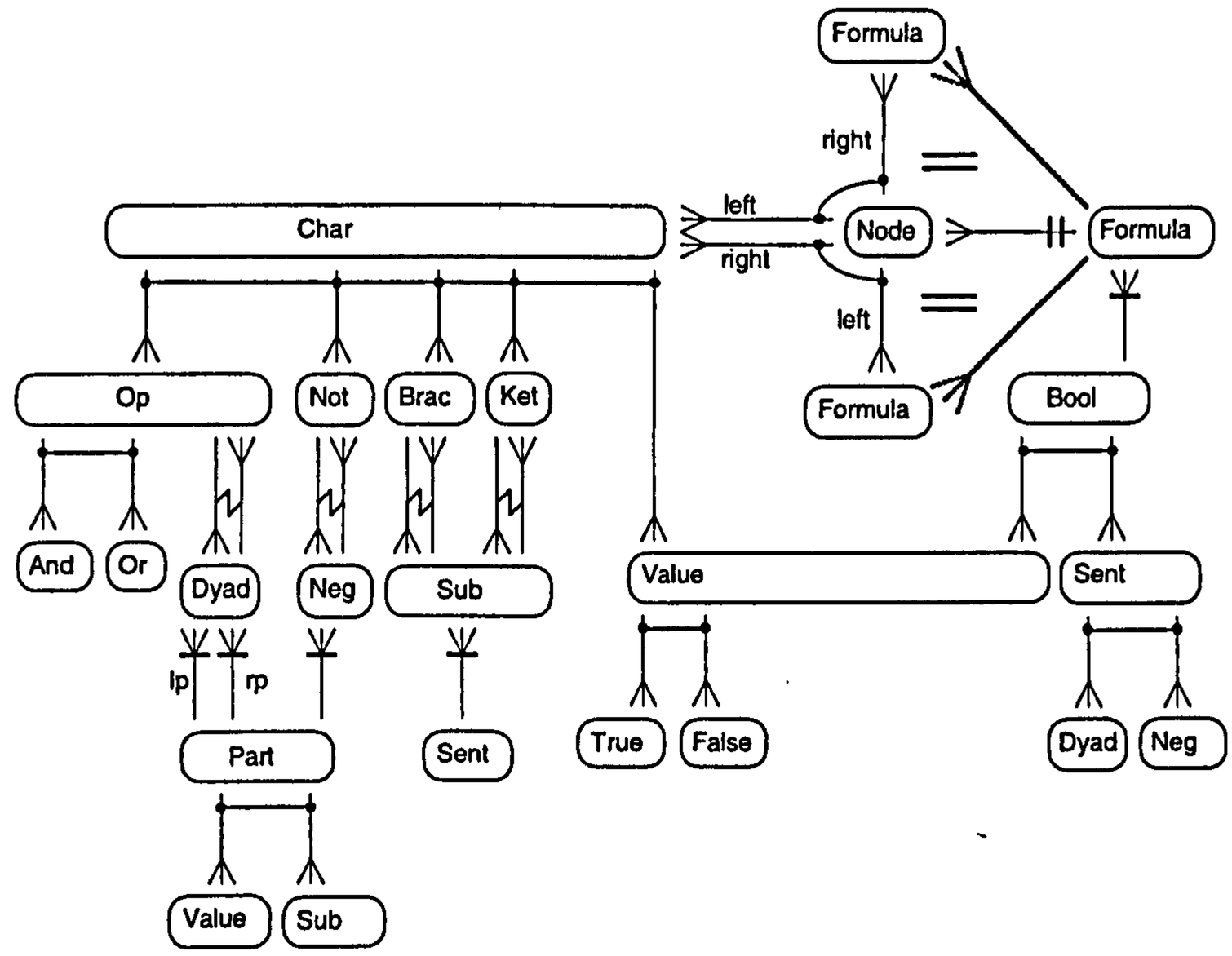


Figure B.2 Constraints on Boolean formulae

The challenge (not taken up here) is to describe in a sketch the full computation of parsing. Apparently in this case the recursion in the context-free grammar has a polynomial bound, and we should be able to carry out the recursion diagrammatically by means of limits and colimits, within the logic of FM-sketches.

Appendix C.

A Logical Proof Concerning a Simple Rewrite Rule

This proof illustrates how sketches can be used for rigorous reasoning about the effects of rewriting operations. We consider the example of generative rewriting given in (§6.2.4.2), of adding branches to trees.

C.1 Two Rules that Generate a Forest

A simple example of generating an expression is found in the case of trees in a forest, in the sense of (§5.3.2.1). It is easy to see that the two simple rewrite-rules R1 and R3 of (§6.2.4.2) suffice to generate a forest of trees – where R1 adds an isolated node as the base of a new tree, and R3 selects any node of a tree and adds a new branch consisting of a new node attached by a new arc, as for example in [fig C.1].

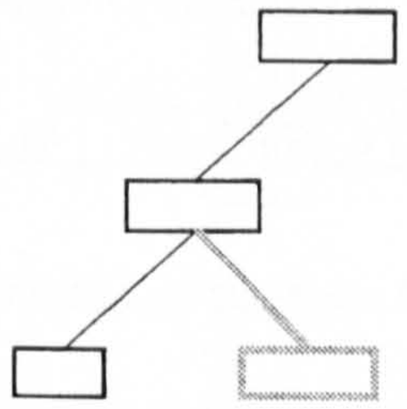


Figure C.1 A single new branch on a lone tree

The soundness of the rules is evidently expressed in the following fact:

> Starting with an empty expression, the application of rules R1 and R3 always results in a syntactic forest.

In the sketch doctrine FM we can prove this assertion and also the property that applying rule R3 leaves the set of trees unchanged. In order to prove completeness – that the two rules can generate *every* finite forest – we would need to represent an inductive argument, which is not considered here.

C.1.1 Proving the Effect of Adding Branches

For purposes of illustration, the proof presented below treats only the properties of R3. It is demonstrated in a diagram-assisted outline that could be expanded into a fully formal presentation. The method analyses the effect of multiple application of rule R3 in parallel – adding a set of branches to the nodes of a set of trees.

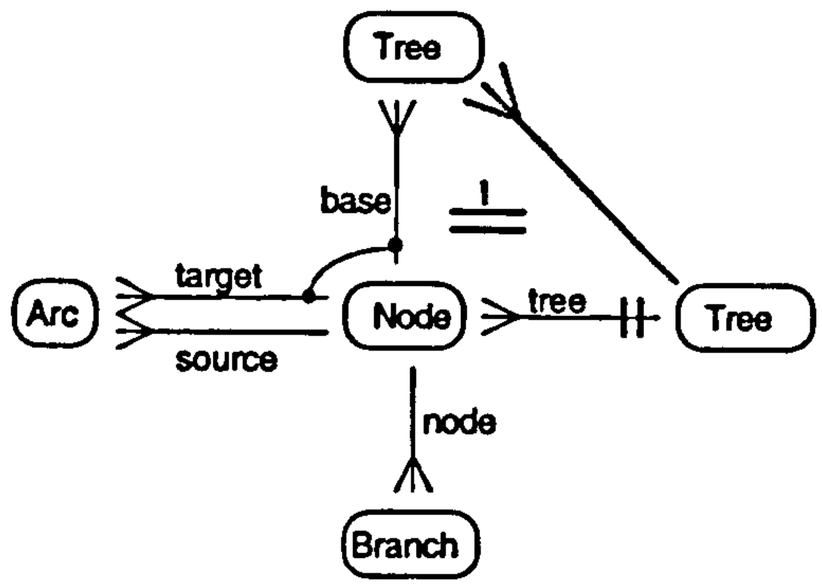


Figure C.2 Sketch FF^+ of a forest with added branches

The operation of adding branches requires a translator from the theory EE of a forest [fig 5.25] to the theory EE^+ of forest-with-added-branches [fig C.2]. This may be carried out by constructing suitable codices between the sketches that present the theories needed.

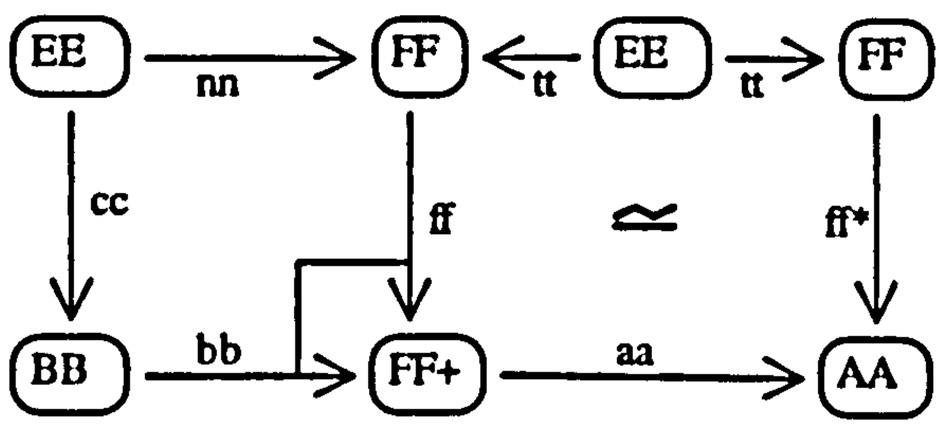


Figure C.3 A meta-sketch of adding branches

The meta-schema [fig C.3] is drawn to depict the constituents of a derived sketch *AA* which is constructed in the proof in order to define the application of the rule. The codex *aa* is a deductive extension of FF^+, and therefore *AA* is a part of the theory EE^+.

In [fig C.3 left], *BB* is the sketch consists of two entities connected by one map – corresponding to the relation between *Node* and *Branch* in FF^+. The codex *bb* is the subsketch of FF^+ :

(Node) —node→ (Branch) whose shape is that of *BB* . The codex *ff* embeds *FF* as a subsketch of FF^+.

The image *bb* is joined to *ff* (using a pushout construction on sketches) to give the sketch FF^+ for a forest-and-new-branches [fig C.2]. The two images *ff*, *bb* share the entity *Node*, to which branches are joined. The sharings are depicted with the help of the sketch *EE* which consists simply of a single entity, with both *nn* and *cc* being the subsketch (Node) , either of *FF* or *BB* respectively.

In [fig C.3 right], the images *ff*, *ff'* share the entity *Tree* – denoting the set of trees of the forest.

The codex *tt* is the subsketch (Tree) within *FF*. The sketch *AA* contains two images (`ff;aa` and `ff*`) of the sketch *FF* for a forest, and an image *bb* of a sketch *BB* for the new branches. The image (`ff;aa`) specifies the forest beforehand, and *ff** yields the forest after the branches have been added. The new forest-schema [fig C.4] for *ff** is shown with derived entities and maps named with an added asterisk. This is just a renaming of *FF*.

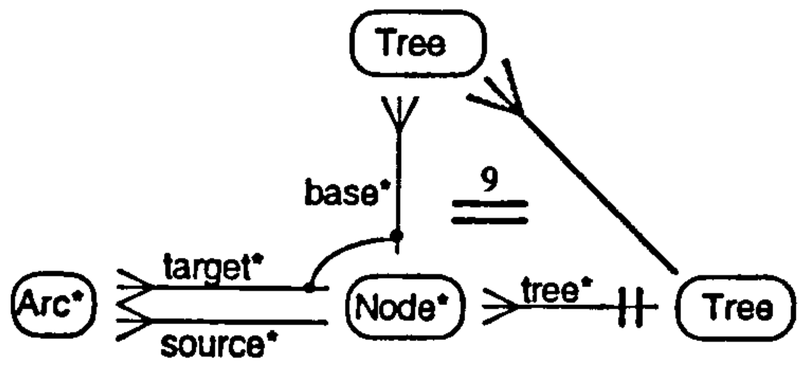


Figure C.4 A new forest *ff**

The correspondence between the old and new tree of the instance [fig C.1] is depicted using the proto-notation described in (§5.4.3.4); in the proto-schema [fig C.5], the tree on the left has a new 'branch' marked, and the resulting new tree copies the old tree and translates the branch into a new arc whose target is a new node. (The branch-attaching map *node* from *Branch* to *Node* is omitted for tidiness).

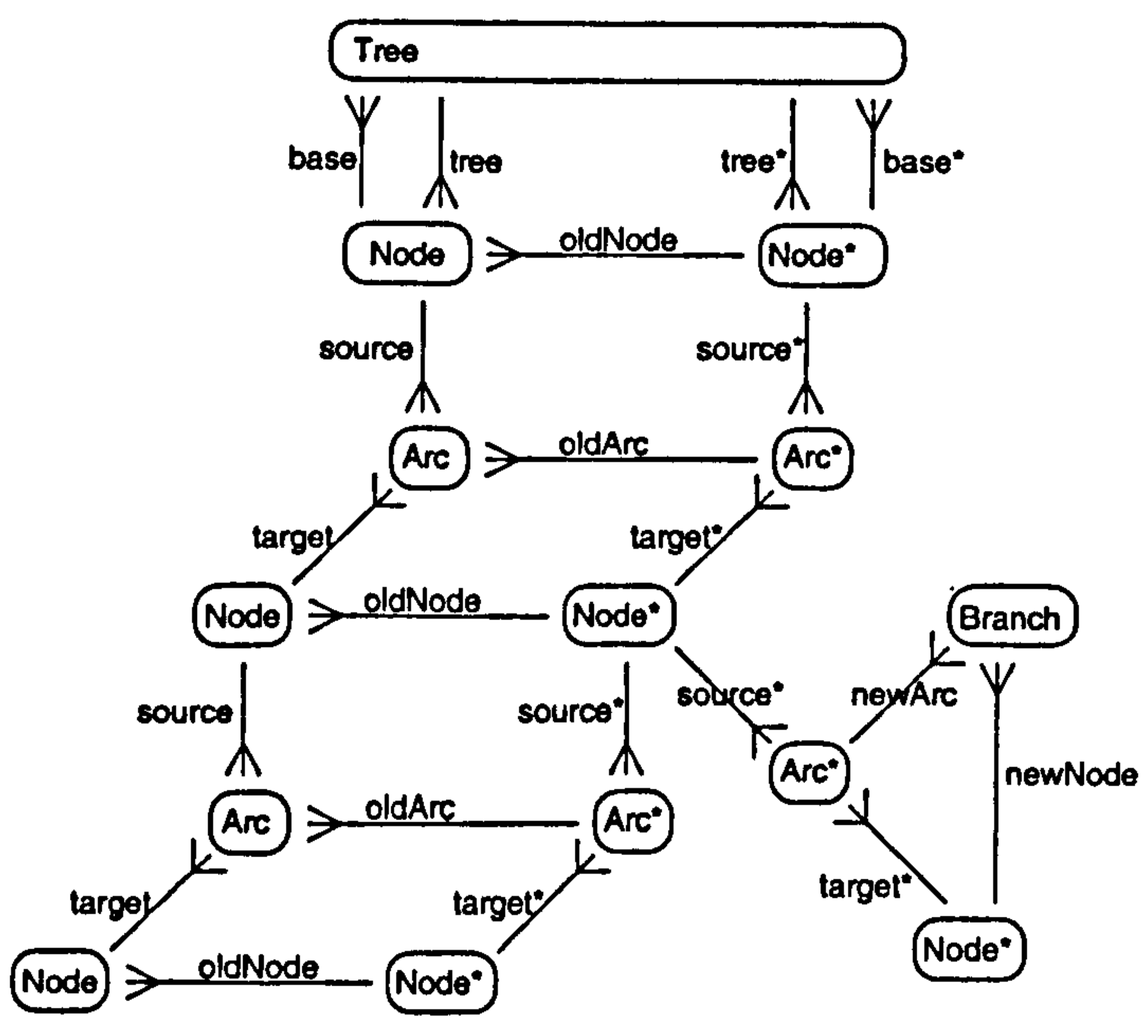


Figure C.5 Relating signatures of old trees, branches and new trees

C.2 The Proof

The proof proceeds in stages. The first step is to construct the new entities and maps of the new forest *ff** of [fig C.4], which expresses the assertions to be proved. Schema [fig C.6] depicts this construction. In order to confirm that the constructions still satisfy the syntax of a forest [fig C.4]

we must justify three assertions: The base of a new tree belongs to that tree itself; *target** and *base** form a disjoint union; trees are the components of the digraph (*source**, *target**).

The schemas supporting the proof depict construction and inferences; derived items are shown in bold, and equalities are numbered for ease of reference.

C.2.1 Constructing the New Forest

To construct the new entities and maps which constitute the new forest, we add new members to the sets of arcs and nodes.

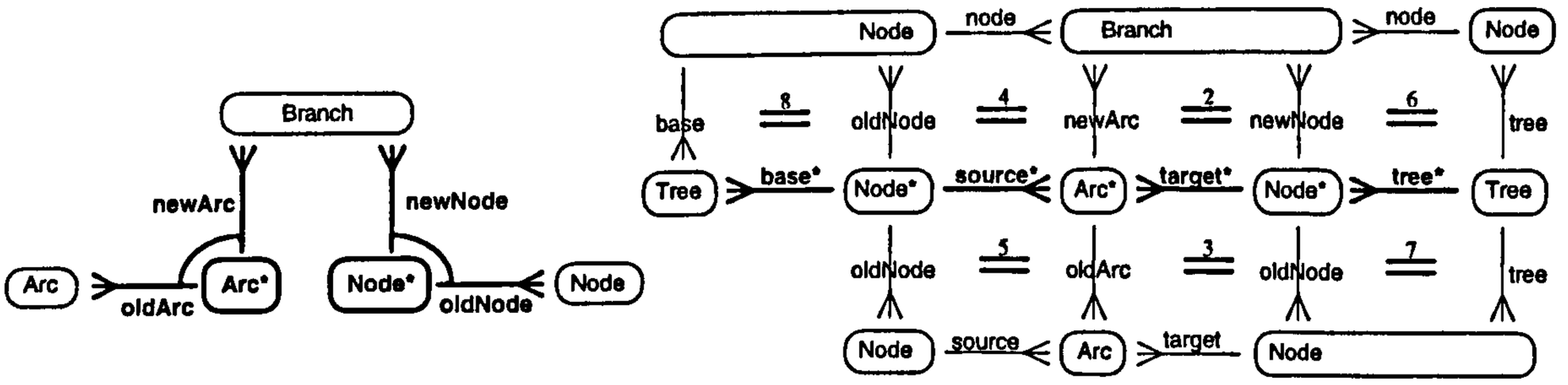


Figure C.6 Stages in interpreting the new trees

New entities ***Arc**** and ***Node**** are defined by adding in a *newArc* and a *newNode* for each branch, by means of disjoint-union constructions.

The ***base**** of any Tree remains as the *oldNode* at its *base*. [E8]

From *Arc**, a map ***source**** is defined by cases:

the *source** of a *newArc* is the *oldNode* at the branching *node*; [E4]

the *source** of an *oldArc* is the *oldNode* at its *source*. [E5]

From *Arc**, a map ***target**** is defined 'by cases' :

the *target** of a *newArc* is a *newNode*; [E2]

the *target** of an *oldArc* is the *target* of an *oldNode*. [E3]

From *Node**, a map ***tree**** is defined by cases:

the *tree** of a *newNode* is the *tree* of the branching *node*; [E6]

the *tree** of an *oldNode* is just its *tree*. [E7]

Schemas [fig 5.25], [fig C.4] and [fig C.6] together show the sketch *AA* referred to in [fig C.3].

C.2.2 Finding the Bases of New Trees

To show that the new base of a tree belongs to that tree itself we need only simple composition of equalities [fig C.7].

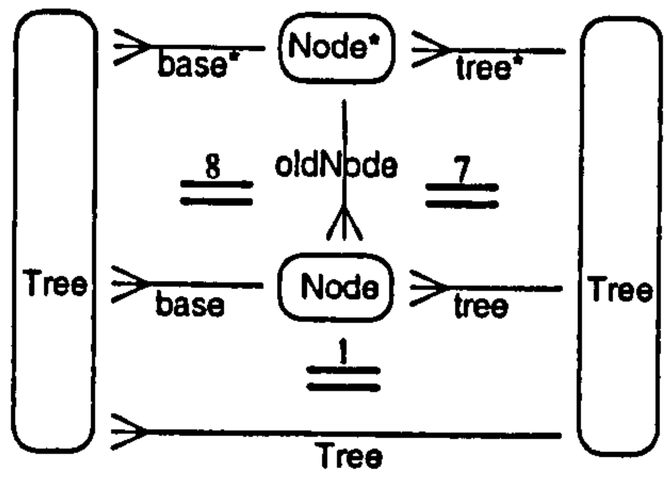


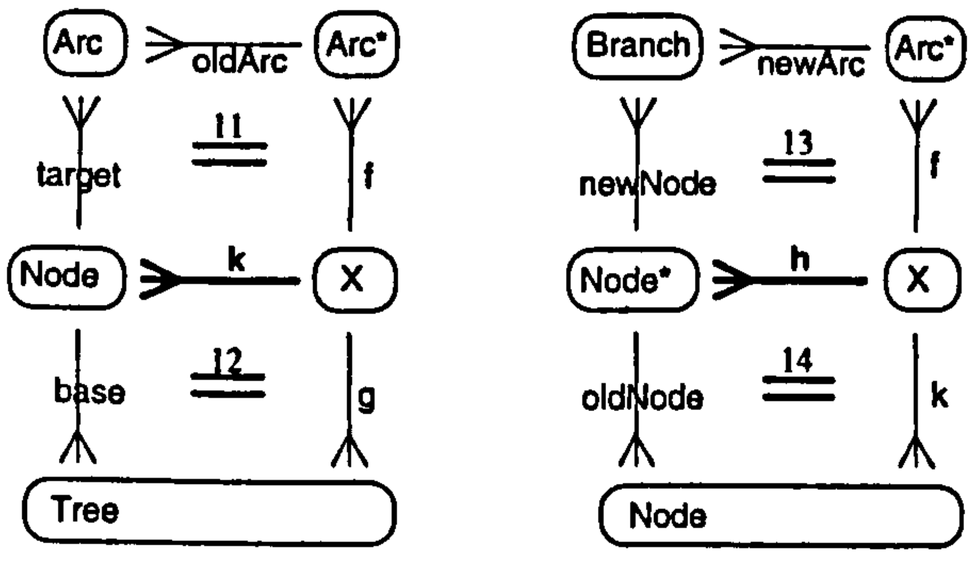
Figure C.7 Proof of `(base*;tree* = Tree)`

To show that `base*;tree* = Tree`. [E9]:-

`base*;tree* = base;oldNode;tree* = base;tree = Tree`

[E8 E7 E1 ⇒ E9].

To show that *target** and *base** form a disjoint union, we require to find an unique map *h* from *Node**, under the circumstances depicted in [fig C.8].



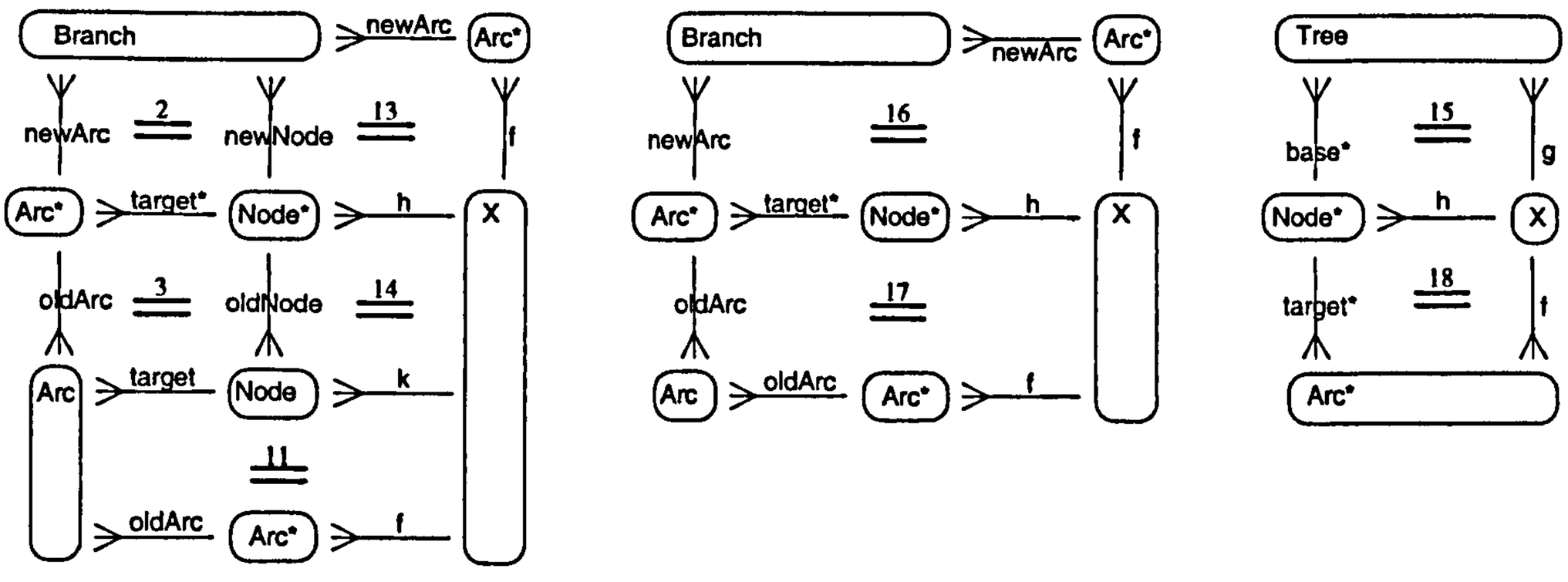


Figure C.8 Proving the existence of h

The colimit property required of *Node** holds if, given any entity *X* and maps *f*, *g* such that:

`f: Arc* → X and g: Tree → X`

then there is an unique map to *X* that converts *target** to *f* and *base** to *g*, i.e.:-

`∃!h:Node* → X • target*;h = f ∧ base*;h = g [E15 E18]`

We can show this by constructing a map *h* as follows, with the help of a map *k*, and then showing it to be unique:-

Because *Node* is a disjoint-union, and `oldArc;f : Arc → X` , we can define unique `k: Node → X`

$\exists!$`k:Node → X • target;k = oldArc;f` $\wedge$ `base;k = g` [E11 E12]

Because *Node** is a disjoint-union, we can define *h* by cases:

Let `newNode;h := newArc;f` and `oldNode;h := k` [E13 E14]

Then `base*;h = base;oldNode;h = base;k = g` [E8 E14 E12 $\Rightarrow$ E15]

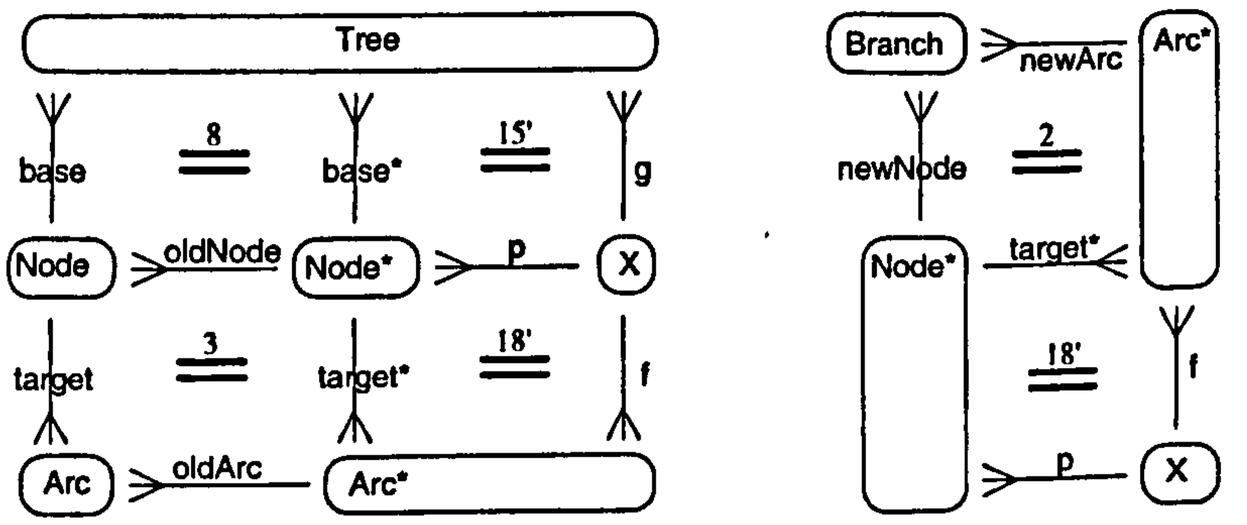
And by cases,

`newArc;target*;h = newNode;h = newArc;f` [E2 E13 $\Rightarrow$ E16]

`oldArc;target*;h = target;oldNode;h = target;k = oldArc;f`
[E13 E14 E11 $\Rightarrow$ E17]

Hence, `target*;h = f` . [E16 E17 $\Rightarrow$ E18]

so that *h* has the required properties [E15 and E18].



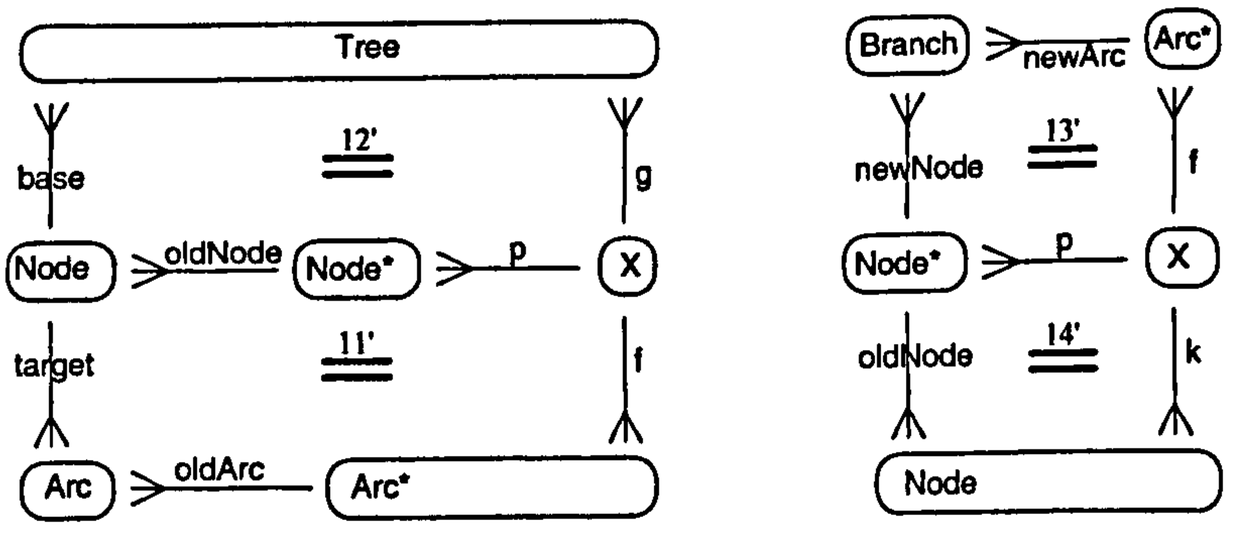


Figure C.9 Proving uniqueness: p = h

For **uniqueness** of *h*, let *p* satisfy `target*;p = f` and `base*;p = g`. [E18' $\wedge$ E15']

Then `base;oldNode;p = base*;p = g` [E8 E15' $\Rightarrow$ E12']

And `target;oldNode;p = oldArc;target*;p = oldArc;f` [E3 E18' $\Rightarrow$ E11']

By definition of *k*, it follows that `oldNode;p = k` [E12' E11' $\Rightarrow$ E14']

Since `newNode;p = newArc;target*;p = newArc;f` [E2 E18' $\Rightarrow$ E13']

By definition of *h*, it follows that `p = h`. [E13' $\wedge$ E14' $\Rightarrow$ p=h]

C.2.2 The New Trees are the Old Trees

To show that no extra trees are created nor existing trees removed, we need to show that trees are the components of the new digraph (*source**, *target**).

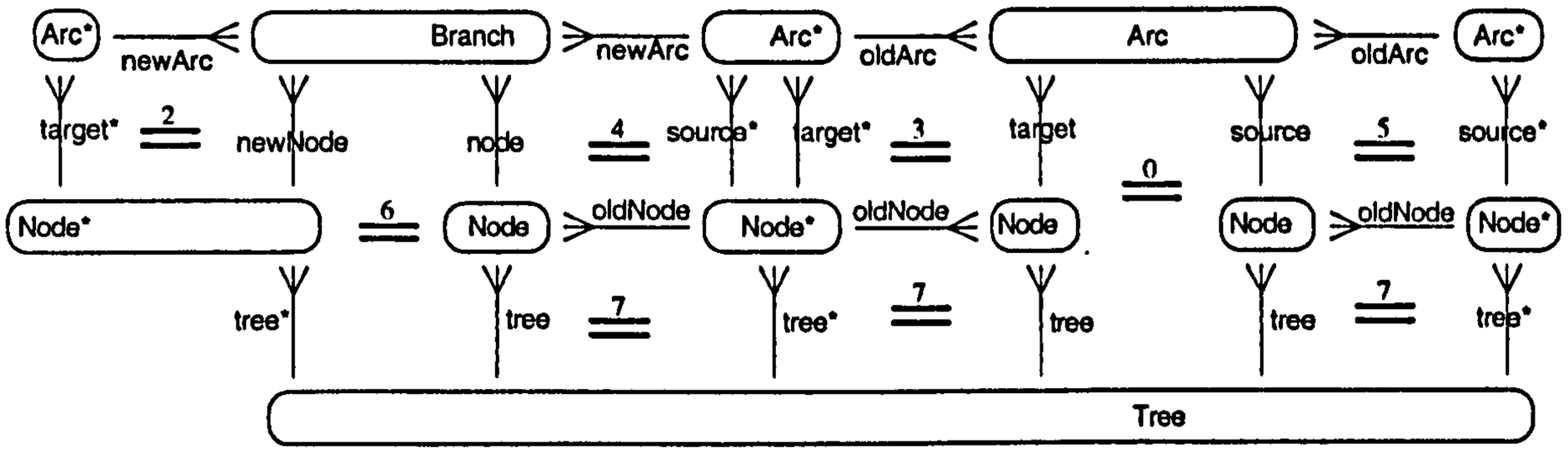


Figure C.10 Proof of E21

To show tree*: Node* → Tree finds components of the graph, we show first that it satisfies the equality: `source*;tree* = target*;tree*` [E21]

Proof by cases [fig C.10]:

`newArc;source*;tree* = node;oldNode;tree* = node;tree`
`= newNode;tree* = newArc; target*;tree*` [E4 E7 E6 E2 ⇒ E19]

`oldArc;source*;tree* = source;oldNode;tree* = source;tree`
`= target;tree = target;oldNode;tree* = oldArc;target*;tree*`
[E5 E7 E0 E7 E3 ⇒ E20]

And it follows that [E19 E20 ⇒ E21] (not depicted).

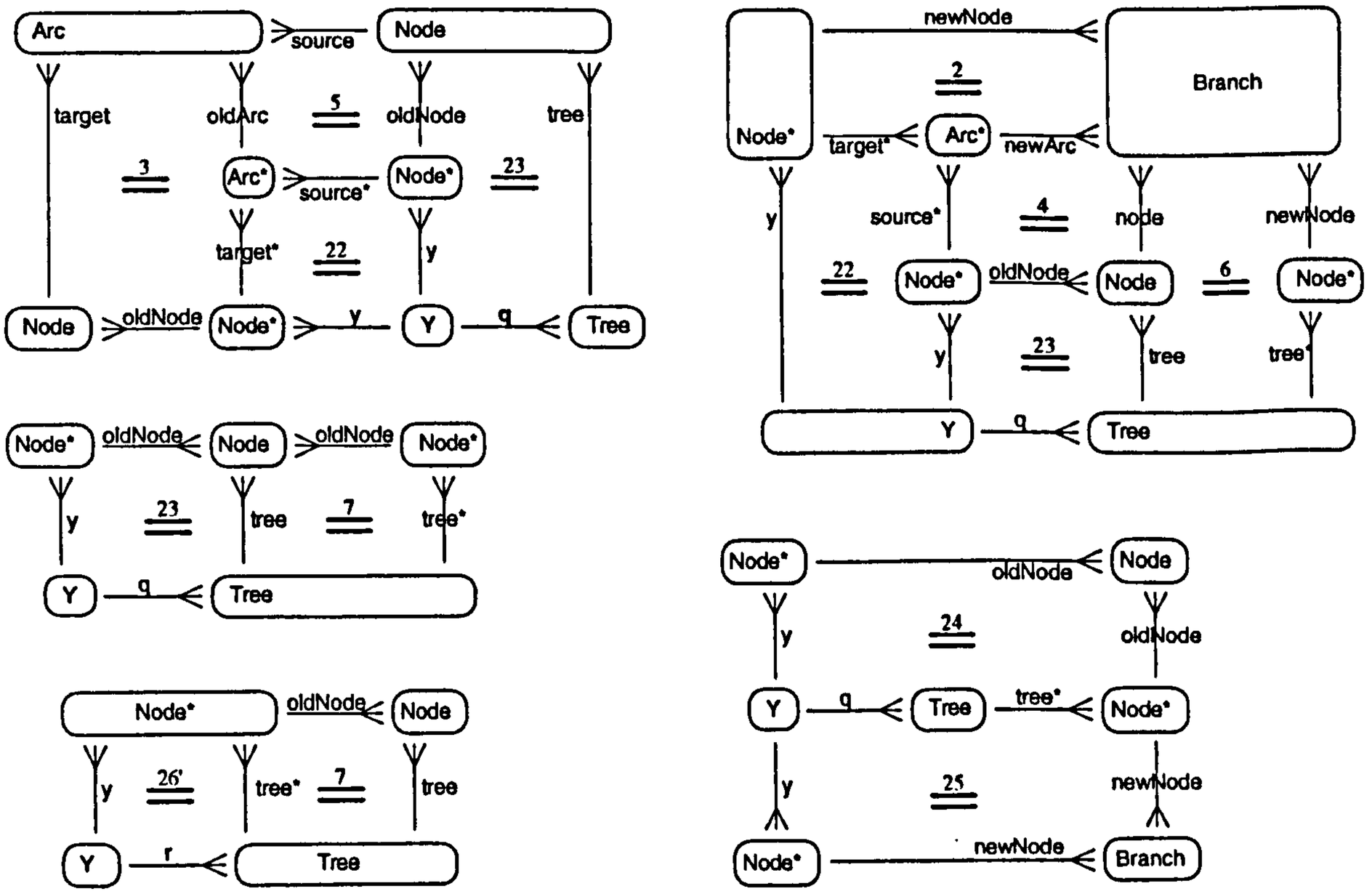


Figure C.11 Proving colimit properties of tree*.

Secondly [fig C.11], we show that for any entity *Y* and map *y* satisfying the same property, y factorizes through *tree**.

Let `y: Node* → Y`

satisfy `source*;y = target*;y` [E22]

Show that there exists `q: Tree → Y` such that `tree*;q = y` [E26]

Now,

`oldArc;source*;y = source;oldNode;y` [E5]

`oldArc;target*;y = target;oldNode;y` [E3]

So we define q uniquely by `tree;q = oldNode;y` [def of tree] [E23]

Then `oldNode;tree*;q = tree;q = oldNode;y` [E7 E23 ⇒ E24]

And `newNode;tree*;q = node;tree;q = node;oldNode;y`

`= newArc;source*;y = newArc;target*;y = newNode;y` [E6 E23 E4 E22 E2 ⇒ E25]

Hence we have the factorization `tree*;q = y` [E24 E25 ⇒ E26]

Thirdly we show that *q* is unique:

Letting `tree*;r = y` [E26'], we show `r = q`.

Immediately, `oldNode;y = oldNode;tree*;r = tree;r` [E7 E26' ⇒ E23']

Hence by uniqueness of *q* in [E23], we have `r = q`.

This completes the proof.

C.3 Remarks

Simple theorems of this kind can be proved more easily as instances of general theorems – in this case concerning colimits in categories. The proof above demonstrates an elementary style of formal reasoning determined by the rules of the doctrine. What is needed is an automated proof-assistant to carry out the mechanics of the process of applying deductive rules and help in searching for a proof.

Appendix D.

A Short Glossary of Terms

For convenience the definitions of new or adapted terminology are listed below, together with reference to the place where each term is introduced.

Codex: A codex is a sketch-morphism – a map between sketches that preserves connective signature and constraints. (§6.1.2)

A codex can *extend* a sketch with extra entities, maps and constraints.

A *deductive* codex extends a sketch within the same theory; all extra entities and maps are constructions on the sketch, and all extra constraints are deductions.

Community of notations: a set of related *logical instruments* that assist visual presentation, reasoning and communication (regarding system designs) within a common environment. (§4.5.1)

Embody: To embody an abstract expression-form is to realize it in some physical medium. (§4.5.1)

Expression: a model of a syntactic sketch.

Concrete expressions are syntactic models in a medium, a category of finite sets.

Abstract expressions are codices to some theory-category.

Form: an underlying combinatoric structure for an expression as defined by a syntactic signature for the notation. Forms are *graphoid*, i.e. graph-like. (§4.4)

The category of forms formalizes the part-whole relation on expressions (§4.4.4).

A *well-formed form* (wff) is a whole expression. (§4.4.1)

The *form-space* is the category of forms (including all those that are not well-formed).

Interpret: To interpret an expression in the subject domain is to recognize it as an action within the cultural world of shared ideas. (§4.5.1)

Interpretant: a semantic entity that corresponds to a graphical sign in an expression. (§4.5.1)

Language: The 'language' of a notation is the category of models of its syntactic sketch.

Medium: a category of (finite) sets and functions in which expressions are modelled.

Meta-sketch: a sketch that is interpreted in the category **Sk** of sketches itself. (§6.1.2)

Meta-schema: a schema that depicts (part of) a meta-sketch.

Model: a codex to a category, or sometimes a functor from a theory to a category. (§5.1.3)

Referent: A referent of a (graphical) token is an actual object or state that is to be found in the whole context that surrounds a displayed expression containing the token. (§4.5.1)

Semiosis: A notation's semiosis is its character and structure as a sign-system. (§4.2)

Situated: An expression is situated when it is embedded in a formal *presupposed* context – i.e. a context where it would be appropriate.

Syntax: The syntax of a notation is the layered coding structure that defines the relation between graphical and semantic forms.

Tagmatic: a layer of syntax in which the entities are types of syntagmatic arrangement of lexical items. (§4.5.1)

Tectonics: The tectonics of a notation is the layered theory that defines its semiosis.

Theory: A formal syntactic theory is a deductively complete class of entities, maps and constraints, generated from a sketch (that presents the theory) by the logical rules of a doctrine. (§6.1.1)

Transform: A transform between models is a family of functions on arranged sets of items. (§6.1.2)

Translator:a functor between theory-categories. (§6.1.2)

A translator from theory A to theory B translates B-expressions as A-expressions.

Appendix E.

A Note on the Sketching of Metaphor Structure

There is enough substance in the theory of Chapter 4 to help understand the mechanisms of metaphor. We consider how this might proceed, by taking the example of data flow.

A Treatment of Metaphor

Let *A* and *S* denote formal theories that define a familiar and an unfamiliar domain respectively. An analogy between the domains *A* and *S* may be represented by a span from some theory *M*, which contains conceptual structures common to both domains.

$$A \leftarrow M \rightarrow S$$

In the example, A refers to fluid flow, and S to data flow. We seek a way of notating *S* that suggests to the viewer the familiar domain *A* in order to motivate understanding of *S*. This is done by providing a visual cue *C* for the familiar domain. The cue is a visual aspect of the domain *A*, such as the linear shape of a duct or pipe. The occurrence of a line on the diagram indicates the route for data to flow. The theory *C* conceptualizes the visual notion of pipe or line (and therefore not the invisible fluid or data that flows in it).

When we notate *S* in syntax *G* via analogy *R*, we enforce semantic well-formedness constraints (from *S*) onto drawings. There are thus two analogies operating on *S*:

$$A \leftarrow M \rightarrow S \leftarrow R \rightarrow G$$

The cue *C* carries graphical properties that are true in *S* by analogy – the shape of a line reflects the attributes of a data channel. The cue also shares these properties with *M*, the metaphorical image of domain *A* – where the shape of a pipe must be that of a duct for fluid.

We wish the cue to be present graphically, which we achieve by extending *C* into a syntactic relation *R* that spans *S* and *G*. As a result, *C* becomes a span between *M* and *R*.

$$M \leftarrow C \rightarrow R$$

A simple equality of paths specifies that *C* describes the same structures in the metaphoric analogy with *S* and in the graphically expressed part of *S*.

Let *C* define a visual part of *A* that serves as a cue.

$$C \rightarrow M \rightarrow S \quad = \quad C \rightarrow R \rightarrow S$$

In order to qualify as a metaphor, the maps from *C* must identify cues in *G* that suggest the familiar domain *A* to the viewer, who must guess at the extent and precision of the map into *A* in order to transfer knowledge of *A* to meanings in *S*. Thus the viewer must guess which attributes of pipes are salient in a discussion of data channels, but also whether other symbols in a diagram bear a similarity of meaning to reservoirs, pumps and desalination units known about in *A* but not in *C*.

Here *M* covers a part of the intended subject domain *S*. If we design the notation with iconism as a guiding principle, we may wish to cover the whole of *S* with overlapping metaphors.

This brief analysis offers a way of structuring and exploring possible analogies and metaphors that may help motivate a new notation.

Appendix F.

Smalltalk *Classes* from the Prototype Implementation

Here is a summary of the prototype as so far constructed. In view of the object-oriented approach, the system is described in terms of its main Classes of Object.

Structure of the Implementation

The principal Object in the application is a *FigureEditor* that allows editing of expressions in any formally specified notation. The editor is applied to a *Figure* that is the expression being edited, displayed in a window called a *FigurePane*, which may be drawn upon by a *Pencil*. A figure consists of a set of items of various sorts (in a Class *ItemSort*). Each *Item* is linked to other items according to the data on sorts for the notation as specified in its *Sketch*. Graphical depiction of these syntactic items is defined in a *Forma*, and details for editing the sketched notation are held in an *Editor* for the forma.

The main Classes and subclasses are:-

FigureEditor (SchemaEditor MetaEditor)
Figure (Schema)
Item (Drawn (Textual) Frame Glyph Peg Site Restraint (Constraint))
Sketch (Forma (Editor))
FigurePane
Pencil

FigureEditor

FigureEditor is a subclass of the Smalltalk system's Class *ViewManager*. A View-Manager maintains an application window and controls user interaction, menu management, opening and closing an editing session. Data for a Figure-Editor record the status during the editing session. These are the *pane* (window) upon which a figure can be drawn, the *pen* or drawing implement being used, the *figure* being edited, and the *editor* that is being used. In the act of editing, there will be the mode of *operation* in force, the *rule* being applied, and the actual details of a *rewrite* in progress. *FigureEditor* has Methods for accessing this status information, and for signalling the operation mode: whether adding/removing an item, adding/removing an annotation, editing, formatting, or applying a rule.

MetaEditor is a subclass of *FigureEditor*. *MetaEditor* has Methods for opening a window for

editing schemas of a target-notation, and for opening figure-editors. It enables the definition of rewrite rules, and the building and modifying of menus.

Figure

A figure is stored as a set of items of various sorts and a forma that specifies syntax and graphics. *Figure* has Methods for adding and deleting items, for finding an item's attachments, for determining specific sets of formatting items, and for saving and retrieving figures stored on disk.

Schema is a subclass of *Figure*, specialized for syntactic SIGN schemas. A schema has Methods for finding subsets of items and for connectivity information special to schemas. A suite of completed schemas is interpreted as a sketch.

Item

An item is an element of a figure, stored as a name identifying what sort of item it is, with a sequence of links to other items. *Item* Methods give access to these data, and allow paths of connectivity to be calculated; items may be encoded for saving on file. Its subclasses are for graphical or other items that have associated actions in the system.

DrawnItem is a subclass of *Item*. A drawn-item is a primitive graphical shape such as a line or circle. It has Methods for drawing itself on screen or on a printer, and for testing if a selected point on screen lies upon itself.

Textual is a subclass of *DrawnItem*, with Methods for editing and printing characters and strings, and for calculating their sizes.

Restraint is a subclass of *Item*. A restraint is a primitive geometric restriction on a vertex, for example restraining it to remain on one side of a given line. It has Methods for checking each kind of restraint and temporarily drawing itself as a link – for use when designing the graphical realization of an item.

Constraint is a subclass of *Restraint*. A geometric constraint determines how the position of a vertex is calculated from a set of vertices that it depends on. *Constraint* Methods control the activation of constraints and calculate the effects of movement.

Sketch

A sketch stores names of entities and maps, a definition of its connectivity as a directed graph, and its set of formal constraints on the graph. *Sketch* Methods provide access to these data, and

to allow a sketch to be built by combining compatible sketches. It has basic facilities to support reasoning.

Forma is a subclass of *Sketch*. A forma stores also the graphical information needed for drawing models of the sketch as adjustable figures in a figure-pane. Its Methods give access to the graphical sorts, maps and constraints [not yet implemented].

Editor is a subclass of *Forma*. An editor stores definitions of all rewrite-rules to be made available during editing expression in the specified notation. It also defines how the rules are offered to the user on visual menus [or via other protocols]. It has Methods for accessing and modifying both the rule-definitions and the menu structure offered.

FigurePane

FigurePane is a subclass of the standard Smalltalk Class *GraphicsMedium*. A figure-pane is a medium (a window) on which a figure is drawn under the direction of a figure-editor. *FigurePane* has Methods for locating the mouse and identifying items selected by the user, and for drawing, hiding, and showing selections, by recolouring highlighted items. It manages format operations, calculating and storing the constraints, restraints, and marks that are affected in formatting. It also controls the retrieving and saving of figures on disk.

Pencil

Pencil is a subclass of *GraphicsTool*, a standard Smalltalk Class. A pencil can draw any item on its pane, by giving control to the item. It has Methods for changing its effect on the pane (e.g. mode or colour), for handling construction lines (rubber bands) and for drawing all graphical primitives.

ALL MISSING
PAGES ARE
BLANK
IN
ORIGINAL